IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

INVENTOR: **Minos Garofalakis et al.**

APPLICATION NO.: **09/595,719** EXAMINER: **Huynh, Cong Lac T**

FILED: **June 16, 2000** GROUP ART UNIT: **2178**

ATTORNEY DOCKET NO.: **Garaofalakis-6-1-36-11-10**

TITLE: **DOCUMENT DESCRIPTOR EXTRACTION METHOD**

37 CFR 1.131 DECLARATION

I, MINOS GAROFALAKIS, declare as follows:

1. I am a co-inventor of United States Patent Application No. 10/075,512 filed on June 16, 2000 (hereinafter, "the Application").

2. On or before October 25, 1999, a paper entitled "XTRACT: A System for Extracting Document Type Descriptors From XML Documents" was submitted to the Intellectual Property Department of Lucent Technologies Inc., the assignee of the Application. This paper, co-authored by the same individuals who are the co-inventors of the Application, described in detail the claimed invention. A copy of this paper is attached hereto as Exhibit A.

3. On or about April 2000, Stephen Weed of the law firm of Synnestvedt & Lechner LLP commenced work on writing the Application. On April 28, 2000 I received a draft of the application, the draft in virtually substantively identical form to the Application. A copy of that draft is attached hereto as Exhibit B.

4. A copy of the present application as filed showing the differences relative to Exhibit B is attached hereto as Exhibit C. It should be apparent from this that Exhibit B is substantively identical to the application as filed and that, therefore, the invention disclosed and claimed in the present application was conceived prior to April 28, 2000.

5. On information and belief, and as verified with a phone conversation with Thomas Onka, Esq. of Synnestvedt & Lechner who reviewed Mr. Weed's files, a final draft of the Application was submitted to Jeffrey Weinick, Esq. of Lucent on May 8, 2000. Mr. Weinick approved the draft with only minor, editorial changes. Subsequent to this approval and the signing of the declaration and assignment documents by me and the other co-inventors, the application was filed on June 16, 2000.

The undersigned, being hereby warned that willful false statements and the like so made are punishable by fine or imprisonment, or both, under 18 U.S.C. §1001, and that such willful false statements may jeopardize the validity of the application or any resulting registration, declares that the facts set forth in this application are true; all statements made of his/her knowledge are true; and all statements made on information and belief are believed to be true.

Dated: 3/14/2005

MINOS GAROFALAKIS
Lucent Technologies Inc.

EXHIBIT A

October 25, 1999 Paper

Paper Number 369

XTRACT: A System for Extracting Document Type Descriptors From XML Documents

Minos Garofalakis   Aristides Gionis*   Rajeev Rastogi   S. Seshadri
Kyuseok Shim
600 Mountain Avenue
Murray Hill, NJ 07974
{minos, rastogi, seshadri, shim}@research.bell-labs.com
gionis@cs.stanford.edu

Abstract

XML is rapidly emerging as the new standard for data representation and exchange on the Web. Unlike HTML, tags in XML documents describe the semantics of the data and not how it is to be displayed. In addition, an XML document can be accompanied by a *Document Type Descriptor* (DTD) which plays the role of a schema for an XML data collection. DTDs contain valuable information on the structure of documents and thus have a crucial role in the efficient storage of XML data, as well as the effective formulation and optimization of XML queries. Despite their importance, however, DTDs are *not mandatory*, and it is frequently possible that documents in XML databases will not have accompanying DTDs. In this paper, we propose XTRACT, a novel system for inferring a DTD schema for a database of XML documents. Since the DTD syntax incorporates the full expressive power of *regular expressions*, naive approaches typically fail to produce concise and intuitive DTDs. Instead, the XTRACT inference algorithms employ a sequence of sophisticated steps that involve: (1) finding patterns in the input sequences and replacing them with regular expressions to generate "general" candidate DTDs, (2) factoring candidate DTDs using adaptations of algorithms from the logic optimization literature, and (3) applying the Minimum Description Length (MDL) principle to find the best DTD among the candidates. The results of our experiments with real-life and synthetic DTDs demonstrate the effectiveness of XTRACT's approach in inferring concise and semantically meaningful DTD schemas for XML databases.

1 Introduction

Motivation and Background. The genesis of the Extensible Markup Language (XML) was based on the thesis that structured documents can be freely exchanged and manipulated, if published in a standard, open format. Indeed, as a corroboration of the thesis, XML today promises to enable a suite of next-generation Web applications ranging from intelligent web searching to electronic commerce.

In many respects, XML data is an instance of *semistructured data* [Abi97]. XML documents comprise hierarchically nested collections of *elements*, where each element can be either atomic (i.e., raw character data) or composite (i.e., a sequence of nested subelements). Further, *tags* stored with elements in an XML document describe the semantics of the data rather than simply specifying how the element is to be displayed (as in HTML). Thus, XML data, like semistructured data, is hierarchically structured and self-describing.

A characteristic, however, that distinguishes XML from semistructured data models is the notion of a *Document Type Descriptor* (DTD) that may optionally accompany an XML document. A document's DTD serves the role of a schema specifying the internal structure of the document. Essentially, a DTD specifies for every element, the *regular expression* pattern that subelement sequences of the element need to conform to. DTDs are critical to realizing the promise of XML as the data representation format that enables free interchange of electronic data (EDI) and integration of related news, products, and services information from disparate data sources. This is because, in the absence of DTDs, tagged documents have little meaning. However, once

*Current Address: Department of Computer Science, Stanford University

the major software vendors and corporations agree on domain-specific standards for DTD formats, it would become possible for inter-operating applications to extract, interpret, and analyze the contents of a document based on the DTD that it conforms to.

In addition to enabling the free exchange of electronic documents through industry-wide standards, DTDs also provide the basic mechanism for defining the structure of the underlying XML data. As a consequence, DTDs play a crucial role in the efficient storage of XML data as well as the formulation, optimization, and processing of queries over a collection of XML documents. For instance, in [SHT$^+$99], DTD information is exploited to generate effective relational schemas, which are subsequently employed to efficiently store and query entire XML documents in a relational database. In [DFS99], frequently occurring portions of XML documents are stored in a relational system, while the remainder is stored in an overflow graph; once again, the DTD is exploited to simplify overflow mappings. Similarly, DTDs can be used to devise efficient plans for queries and thus speed up query evaluation in XML databases by restricting the search to only relevant portions of the data (see, for example, [GW97, FS97]). The basic idea is to use the knowledge of the structure of the data captured by the DTD to prune elements that cannot potentially satisfy the path expression in the query. Finally, by shedding light on how the underlying data is structured, DTDs aid users in forming meaningful queries over the XML database.

Despite their importance, however, DTDs are *not mandatory* and an XML document may not always have an accompanying DTD. In fact, several recent papers (e.g., [GMW99, Wid91]) claim that it is frequently possible that only specific portions of XML databases will have associated DTDs, while the overall database is still "schema-less". This may be the case, for instance, when large volumes of XML documents are automatically generated from data stored in relational databases, flat files (e.g., HTML pages, bibliography files), or other semistructured data repositories. Since very little data is in XML format today, it is very likely that, at least initially, the majority of XML documents will be automatically generated from pre-existing data sources by a new generation of software tools. In most cases, such automatically-created document collections will not have an accompanying DTD.

Therefore, based on the above discussion on the virtues of a DTD, it is important to devise algorithms and tools that can infer an accurate, meaningful DTD for a given collection of XML documents (i.e., *instances* of the DTD). This is *not* an easy task. Since the DTD syntax incorporates the full specification power of regular expressions, manually deducing such a DTD schema for even a small set of XML documents created by a user could prove to be a process of daunting complexity. Furthermore, as we show in this paper, naive approaches fail to deliver meaningful and intuitive DTD descriptions of the underlying data. Both problems are, of course, exacerbated for *large* XML document collections. In light of the several benefits of DTDs, we can motivate a myriad of potential applications for efficient, automated DTD discovery tools. For example, users or domain experts looking for a meaningful description of their XML data can use the DTD description returned by such tools as a starting point from which more refined schemas can be generated. As another application, consider an employment web site that integrates information on job openings from thousands of different web sites including company home pages, newspaper classified sites, and so on. These XML documents, although related, may not all have the same structure and, even if some of the documents are accompanied by DTDs, the DTDs may not be identical. An alternative to manually transforming all the XML documents to conform to a single format would be to simply store the documents in their original formats and use DTD discovery tools to derive a single DTD description for the entire database. This inferred DTD can then help in the formulation, optimization, and processing of queries over the database of stored XML documents. Finally, the ability to extract DTDs for a range of XML formats supported by the major participants in a specific industrial setting can also aid in the DTD standardization process for the industry.

Our Contributions. In this paper, we describe the architecture of XTRACT, a novel system for inferring an accurate, meaningful DTD schema for a repository of XML documents. A naive and straightforward solution to our DTD extraction problem would be to infer as the DTD for an element, a "concise" expression which describes *exactly* all the sequences of subelements nested within the element in the entire document collection. As we demonstrate in Section 3, however, the DTDs generated by this approach tend to be voluminous and unintuitive (especially for large XML document collections). In fact, we discover that accurate and meaningful DTD schemas that are also intuitive and appealing to humans (i.e., resemble what a human expert is likely to come up with) tend to *generalize*. That is, "good" DTDs are typically regular expressions describing subelement sequences that

may not actually occur in the input XML documents. (Note that this, in fact, is always the case for DTD regular expressions that correspond to infinite regular languages, e.g., DTDs containing one or more Kleene stars "*" [HU79].) In practice, however, there are numerous such candidate DTDs that generalize the subelement sequences in the input, and choosing the DTD that best describes the structure of these sequences is a non-trivial task. In the inference algorithms employed in the XTRACT system, we propose the following novel combination of sophisticated techniques to generate DTD schemas that effectively capture the structure of the input sequences.

- **Generalization.** As a first step, the XTRACT system employs novel heuristic algorithms for finding patterns in each input sequence and replacing them with appropriate regular expressions to produce more general candidate DTDs. The main goal of the generalization step is to judiciously introduce metacharacters (like Kleene stars "*") to produce regular subexpressions that generalize the patterns observed in the input sequences. Our generalization heuristics are based on the discovery of frequent, neighboring occurrences of subsequences and symbols within each input sequence. In their effort to introduce a sufficient amount of generalization while avoiding an explosion in the number of resulting patterns, our techniques are inspired by practical, real-life DTD examples.

- **Factoring.** As a second step, the XTRACT system *factors* common subexpressions from the generalized candidate DTDs obtained from the generalization step, in order to make them more concise. The factoring algorithms applied are appropriate adaptations of techniques from the logic optimization literature [BM82, Wan89].

- **Minimum Description Length (MDL) Principle.** In the final and most important step, the XTRACT system employs Rissanen's *Minimum Description Length* (MDL) principle [Ris78, Ris89] to derive an elegant mechanism for composing a near-optimal DTD schema from the set of candidate DTDs generated by the earlier two steps. (Our MDL-based notion of optimality will be defined formally later in the paper.) The MDL principle has its roots in information theory and, essentially, provides a principled, scientific definition of the optimal "theory/model" that can be inferred from a set of data examples [QR89b]. Abstractly, in our problem setting, MDL ranks each candidate DTD depending on the number of bits required to describe the input collection of sequences *in terms of the DTD* (DTDs requiring fewer bits are ranked higher). As a consequence, the optimal DTD according to the MDL principle is the one that is general enough to cover a large subset of the input sequences but, at the same time, captures the structure of the input sequences with a fair amount of detail, so that they can be described easily (with few additional bits) using the DTD. Thus, the MDL principle provides a formal notion of "best DTD" that exactly matches our intuition. Using MDL essentially allows XTRACT to control the amount of generalization introduced in the inferred DTD in a principled, scientific and, at the same time, intuitively appealing fashion.

  We demonstrate that selecting the optimal DTD based on the MDL principle has a direct and natural mapping to the *facility location problem* (FLP), which is known to be NP-complete [Hoc82]. Fortunately, efficient approximation algorithms with guaranteed performance ratios have been proposed for the FLP in the literature [CG99], thus allowing us to efficiently compose the final DTD in a near-optimal manner.

We have implemented our XTRACT DTD derivation algorithms and conducted an extensive experimental study with both real-life and synthetic DTDs. Our findings show that, for a set of random inputs that conform to a predetermined DTD, XTRACT always produces a DTD that is either identical or very close to the original DTD. We also observe that the quality of the DTDs returned by XTRACT is far superior compared to those output by the IBM alphaworks[1] DDbE (Data Descriptors by Example) DTD extraction tool, which is unable to identify a majority of the DTDs. Further, a number of the original DTDs correctly inferred by XTRACT contain several regular expressions terms, some nested within one another. Thus, our experimental results clearly demonstrate the effectiveness of XTRACT's methodology for deducing fairly complex DTDs.

Several extensions to DTDs, e.g., Document Content Descriptors (DCDs) and XML Schemas, are being evolved by the Web community. These extensions aim to add typing information since DTDs treat all data as strings. Therefore, XTRACT, can be

[1] See http://www.alphaworks.ibm.com/formula/xml.

used with little or no changes for inferring DCDs and XML Schemas in conjunction with other mechanisms for inferring the types. However, these proposals are still evolving and none of them have stabilized – therefore, we do not concentrate on these extensions in this paper.

Roadmap. The remainder of the paper is organized as follows. After discussing related work in Section 2, we present an overview of our approach to inferring DTDs in Section 3. Section 4 describes how the MDL principle is employed within XTRACT to compose a "good" DTD from an input set of candidate DTDs. In Sections 5 and 6, we present generalization and factoring algorithms for producing candidate DTDs that are input to the MDL module of XTRACT. Section 7 discusses the results of our experiments with real-life and synthetic DTDs. Finally, we offer concluding remarks in Section 8.

2 Related Work

The problem of mining DTDs from a collection of XML documents, to the best of our knowledge, is novel and has not been previously addressed in the literature. A few DTD extraction software tools can be found on the Web (e.g., the IBM alphaworks DDbE product) – however, it has been our experience that these tools are somewhat naive in their approach and the quality of the DTDs inferred by them is poor (see Section 7).

The problem of extracting a schema from semistructured data has been addressed in [NAM98, GW97, FS97]. Although, XML can be viewed as an instance of semistructured data, the kinds of schema considered in [NAM98, GW97, FS97] are very different from a DTD. The schema extracted by [NAM98, GW97, FS97] attempt to find a typing for semistructured data. Assuming a graph-based model for semistructured data (nodes denote objects and labels on edges denote relationships between them), finding a typing is tantamount to grouping objects that have similarly labeled edges to and from similarly typed objects. The typing then describes this grouping in terms of the labels of the edges to (from) this type of objects and the types of the objects at the other end of the edge. In contrast, one can perhaps view the DTD as having already grouped all objects based on their incoming edges (tag of the element) into the same type and then describing the possible sequence of outgoing edges (subelements) as a regular expression. It is the fact that the outgoing edges from a type can be described by an arbitrary regular expression that distinguishes DTDs from the schemas in semistructured databases. Since the schemas in semistructured databases are expressed using plain sequences or sets of edges, they cannot be used to infer DTDs corresponding to arbitrary regular expressions.

Inference of formal languages from examples has a long and rich history in the field of computational learning theory, and more related to our work is the extensive study of the inference of *DFA*s (deterministic finite automata) [Gol67, Gol78, Ang78] (see also [Pit89] for a detailed survey of the topic). The above line of work is purely theoretical and it focuses on investigating the computational complexity of the language inference problem, while we are mainly interested in devising practical algorithms for real world applications. In this sense, our research is more closely related to the work in [Bra93] which addressed the problem of approximating *roughly equivalent* regular expressions from a long enough string, and the work in [KMU95] where the *MDL* principle was used to infer a *pattern language* from positives examples. However, the problem tackled in [KMU95] is much simpler than ours since they assume that the set of simple patterns whose subset is to be computed is available. Furthermore, the patterns they consider are simple sequences that are permitted to contain single symbol wildcards. In our problem setting, unlike [KMU95], patterns are general regular expressions and are not known apriori.

3 Problem Formulation and Overview of our Approach

In this section, we present a precise definition of the problem of inferring a DTD from a collection of XML documents and then present an overview of the steps performed by the XTRACT system. But first, we present a brief overview of XML and DTDs in the following subsection to make the subsequent discussion concrete.

```
<article>
   <title> A Relational Model for Large Shared Data Banks </title>
   <author>
      <name> E. F. Codd </name>
      <affiliation> IBM Research </affiliation>
   </author>
</article>
```

Figure 1: An Example XML Document

```
<!ELEMENT article(title, author*)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author(name, affiliation)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT affiliation (#PCDATA)>
```

Figure 2: An Example DTD

3.1 Overview of XML and DTDs

An XML document, like an HTML document, consists of nested element structures starting with a root element. Subelements of an element can either be elements or simply character data. Figure 1 illustrates an example XML document, in which the root element (`article`) has two nested subelements (`title` and `author`), and the `author` element in turn has two nested subelements. The `title` element contains character data denoting the title of the article while the name element contains the name of the author of the article. The ordering of subelements within an element is significant in XML. Elements can also have zero or more attribute/value pairs that are stored within the element's start tag. More details on the XML specification can be found in [BPSM].

A DTD is a grammar for describing the structure of an XML document. A DTD constrains the structure of an element by specifying a regular expression that its subelement sequences have to conform to. Figure 2 illustrates a DTD that the XML document in Figure 1 conforms to. The DTD declaration syntax uses commas for sequencing, | for (exclusive) or, parenthesis for grouping and the meta-characters ?,* ,$^+$ to denote, respectively, zero or one, zero or more, and one or more occurrences of the preceding term. As a special case, the DTD corresponding to an element can be ANY which allows an arbitrary XML fragment to be nested within the element. The DTD can also be used to specify the attributes for an element (using the `<!ATTLIST >` declaration) and to declare an attribute that refers to another element (via an IDREF field). We must point out that real-life DTDs can get fairly complex and can sometimes contain several regular expressions terms with multiple levels of nesting (e.g., $((ab)^*c)^*$). We present examples of real-life DTDs in sections 5 and 7.

For brevity, in the remainder of the paper, we denote elements of an XML document by a single letter from the lower case alphabet. Also, we do not include explicit commas in element sequences and regular expressions since they can be inferred in a straightforward fashion.

3.2 Problem Definition

Our primary focus in this paper is to infer a DTD for a collection of XML documents. Thus, for each element that appears in the XML documents, our goal is to derive a regular expression that subelement sequences for the element (in the XML documents) conform to. Note that an element's DTD is completely independent of the DTD for other elements, and only restricts the sequence of subelements nested within the element. Therefore, for simplicity of exposition, in the rest of the paper, we concentrate on the

problem of extracting a DTD for a single element. In this paper, we do not address the problem of computing attribute lists for an element – since these are simple lists, their computation is not particularly challenging.

Let e be an element that appears in the XML documents for which we want to infer the DTD. It is straightforward to compute the sequence of subelements nested within each $< e >< /e >$ pair in the XML documents. Let I denote the set of N such sequences, one sequence for every occurrence of element e in the data. The problem we address in this paper can be stated as follows.

Problem Statement. Given a set I of N input sequences nested within element e, compute a DTD for e such that every sequence in I conforms to the DTD. □

As stated, an obvious solution to the problem is to find the most "concise" regular expression R whose language is I. One mechanism to find such a a regular expression is to factor as much as possible, the expression corresponding to the *or* of sequences in I. Factoring a regular expression makes it "concise" without changing the language of the expression. For example, $ab|ac$ can be factored into $a(b|c)$. An alternate method for computing the most concise regular expression is to first find the automaton with the smallest number of states that accepts I and then derive the regular expression from the automaton (note that the obtained regular expression, however, may not be the shortest regular expression for I). In any case, such a concise regular expression whose language is I, is unfortunately not a "good" DTD in the sense it tends to be voluminous and unintuitive. We illustrate this using the DTD of Figure 2. Suppose we have a collection of XML documents that conform to this DTD. Abbreviating the `title` tag by t, and the `author` tag by a, it is reasonable to expect the following sequences to be the subelement sequences of the `article` element in the collection of XML documents: t, ta, taa, $taaa$, $taaaa$. Clearly, the most concise regular expression for the above language is $t|t(a|a(a|a(a|aa)))$ which is definitely much more voluminous and lot less intuitive than a DTD such as ta^*.

In other words, the obvious solution above never "generalizes" and would therefore never contain metacharacters like $*$ in the inferred DTD. Clearly, a human being would at most times want to use such metacharacters in a DTD to succinctly convey the constraints he/she wishes to impose on the structure of XML documents. Thus, the challenge is to infer for the set of input sequences I, a "general" DTD which is similar to what a human would come up with. However, as the following example illustrates, there can be several possible "generalizations" for a given set of input sequences and thus we need to devise a mechanism for choosing the one that best describes the sequences.

Example 3.1 Consider $I = \{ab, abab, ababab\}$. A number of DTDs match sequences in I – (1) $(a \mid b)^*$, (2) $ab \mid abab \mid ababab$, (3) $(ab)^*$, (4) $ab \mid ab(ab \mid abab)$, and so on. DTD (1) is similar to ANY in that it allows any arbitrary sequence of as and bs, while DTD (2) is simply an *or* of all the sequences in I. DTD (4) is derived from DTD (2) by factoring the subsequence ab from the last two disjuncts of DTD (2). The problem with DTD (1) is that it represents a gross over-generalization of the input, and the inferred DTD completely fails to capture any structure inherent in the input. On the other hand, DTDs (2) and (4) accurately reflect the structure of the input sequences but do not generalize or learn any meaningful patterns which make the DTDs smaller or simpler to understand. Thus, none of the DTDs (1), (2) or (4) seem "good". However, of the above DTDs, (3) has great intuitive appeal since it is succinct and it generalizes the input sequences without losing too much information about the structure of the input sequences. □

Based on the discussion in the above example, we can characterize the set of desirable DTDs by placing the following two qualitative restrictions on the inferred DTD.

R1: The DTD should be concise (i.e., small in size).

R2: The DTD should be precise (i.e, not cover too many sequences not contained in I).

Restriction R1 above ensures that the inferred DTD is easy to understand and succinct, thus eliminating, in many cases, concise regular expressions whose language is I. Restriction R2, on the other hand, attempts to ensure that the DTD is not too general and

captures the structure of input sequences, thus eliminating a DTD such as ANY. While the above restrictions seem reasonable at an intuitive level, there is a problem with devising a solution based on the above restrictions. The problem is that restrictions R1 and R2 conflict with each other. In our earlier example, restriction R1 would favor DTDs (1) and (3), while these DTDs would not be considered good according to criterion R2. The situation is exactly the reverse when we consider DTDs (2) and (4). Thus, in general, there is a tradeoff between a DTD's "conciseness" and it's "preciseness", and a good DTD is one that strikes the right balance between the two. The problem here is that conciseness and preciseness are qualitative notions – in order to resolve the tradeoff between the two, we need to devise quantitative measures for mathematically capturing the two qualitative notions.

3.3 Using the MDL Principle to Define a Good DTD

We use the MDL principle [Ris78, Ris89] to define an information-theoretic measure for quantifying and thereby resolving the tradeoff between the conciseness and preciseness properties of DTDs. The MDL principle has been successfully applied in the past in a variety of situations ranging from constructing good decision tree classifiers [QR89a, MRA95] to learning common patterns in sets of strings [KMU95].

Roughly speaking, the MDL principle states that the best theory to infer from a set of data is the one which minimizes the sum of

(A) the length of the theory, in bits, and

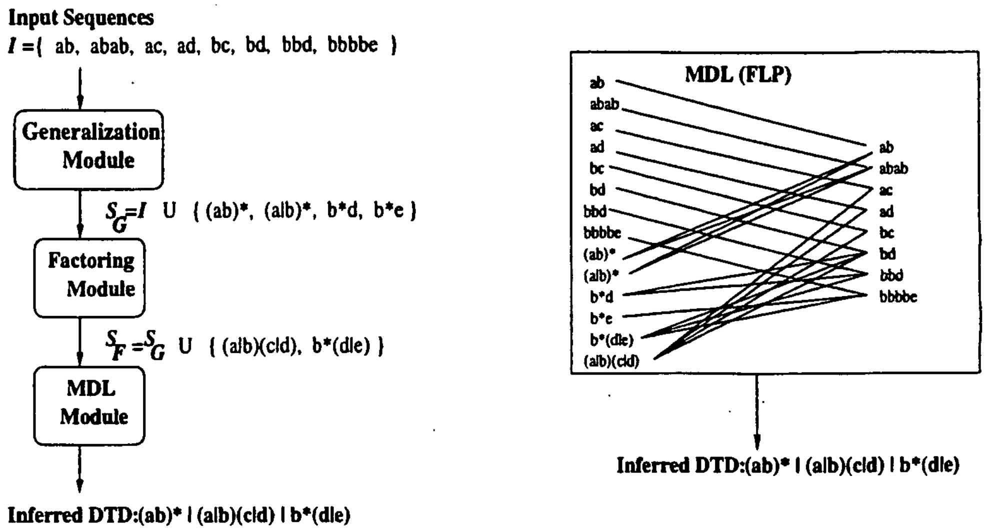
(B) the length of the data, in bits, when encoded with the help of the theory.

We will refer to the above sum, for a theory, as the *MDL cost* for the theory. The MDL principle is a general one and needs to be instantiated appropriately for each situation. In our setting, the theory is the DTD and the data is the sequences in I. Thus, the MDL principle assigns each DTD an MDL cost and ranks the DTDs based on their MDL costs (DTDs with lower MDL costs are ranked higher). Furthermore, parts (A) and (B) of the MDL cost for a DTD depend directly on its conciseness and preciseness, respectively. Part (A) is the number of bits required to describe the DTD and is thus a direct measure of its conciseness. Further, since a DTD that is more precise captures the structure of the input sequences more accurately, fewer bits are required to describe the sequences in I in terms of a more precise DTD. As a result, Part (B) of the MDL cost captures a DTD's preciseness. The MDL cost for a DTD thus provides us with an elegant and principled mechanism (rooted in information theory) for quantifying (and combining) the conflicting concepts of conciseness and preciseness in a single unified framework, and in a manner that is consistent with our intuition. By favoring concise and precise DTDs, and penalizing those that are'nt, it ranks highly exactly those DTDs that would be deemed desirable by humans.

Note that the actual encoding scheme used to specify a DTD as well as the data (with the help of the DTD) plays a critical role in determining the actual values for the two components of the MDL cost. We defer the details of the actual encoding scheme to Section 4. However, in the following example, we employ a simple encoding scheme (a coarser version of the scheme in Section 4) to illustrate how ranking DTDs based on their MDL cost closely matches our intuition of their goodness.

Example 3.2 Consider the input set I and DTDs from Example 3.1. We compute the MDL cost of each DTD, which, as mentioned earlier, is the cost of encoding the DTD itself and the sequences in I in terms of the DTD. We then rank the DTDs based on their MDL costs (DTDs with smaller MDL costs are considered better). In our simple encoding scheme, we assume a cost of 1 unit for each character.

DTD (1), $(a \mid b)^*$, has a cost of 6 for encoding the DTD. In order to encode the sequence *abab* using the DTD, we need one character to specify the number of repetitions of the the term $(a \mid b)$ that precedes the * (in this case, this number is 4), and 4 additional characters to specify which of a or b is chosen from each repetition. Thus, the total cost of encoding *abab* using $(a \mid b)^*$ is 5 and the MDL cost of the DTD is $6 + 3 + 5 + 7 = 21$. Similarly, the MDL cost of DTD (2) can be shown to be 14 (to encode the DTD) + 3 (to encode the input sequences; we need one character to specify the position of the disjunct for each sequence) = 17. The cost of DTD (3) is 5 (to encode the DTD) + 3 (to encode the input sequences – note that we only need to



(a) (b)

Figure 3: Architecture of the XTRACT System

specify the number of repetitions of the term ab for each sequence) = 8. Finally, DTD (4) has a cost of 14 + 5 (1 character to encode sequence ab and 2 characters for each of the other two input sequences) = 19.

Thus, since DTD (3) has the least MDL cost, it would be considered the best DTD by the MDL principle – which matches our intuition. □

From the above example, it follows that the MDL principle indeed provides an elegant mechanism for quantifying and resolving the tradeoff between the conciseness and preciseness properties of DTDs. Specifically,

1. Part (A) of the MDL cost includes the number of bits required to encode the DTD – this ensures that the inferred DTD is succinct.

2. Part (B) of the MDL cost includes the number of bits needed for encoding the input sequences using the DTD. Usually, expressing data in terms of a more general DTD (e.g., $(a \mid b)^*$ in Example 3.2) requires more bits than describing data in terms of a more specific DTD (e.g., $(ab)^*$ in Example 3.2). As a result, using the MDL principle ensures that the DTD we choose is a fairly tight characterization of the data.

The MDL principle, thus, enables us to choose a DTD that strikes the right balance between conciseness and preciseness.

3.4 Overview of the XTRACT System

The architecture of the XTRACT system is illustrated in Figure 3(a). As shown in the figure, the system consists of three main components: the generalization module, the factoring module and the MDL module. Input sequences in I are processed by the three subsystems one after another, the output of one subsystem serving as input to the next. We denote the outputs of the generalization and factoring modules by $\mathcal{S}_\mathcal{G}$ and $\mathcal{S}_\mathcal{F}$, respectively. Observe that both $\mathcal{S}_\mathcal{G}$ and $\mathcal{S}_\mathcal{F}$ contain the initial input sequences in I. This is to ensure that the MDL module has a wide range of DTDs to choose from that includes the obvious DTD which is simply an *or* of all the input sequences in I. In the following, we provide a brief description of each subsystem; we defer a more detailed description of the algorithms employed by each subsystem to later sections.

The Generalization Subsystem. For each input sequence, the generalization module generates zero or more candidate DTDs that are derived by replacing patterns in the input sequence with regular expressions containing metacharacters like * and | (e.g.,

$(ab)^*$, $(a \mid b)^*$). Note that the initial input sequences do not contain metacharacters and so the candidate DTDs introduced by the generalization module are more general. For instance, in Figure 3(a), sequences *abab* and *bbbe* result in the more general candidate DTDs $(ab)^*$, $(a \mid b)^*$ and b^*e to be output by the generalization subsystem. Also, observe that each candidate DTD produced by the generalization module may cover only a subset of the input sequences. Thus, the final DTD output by the MDL module may be an *or* of multiple candidate DTDs.

Ideally, in the generalization phase, we should consider all DTDs that cover one or more input sequences as candidates so that the MDL step can choose the best among them. However, the number of such DTDs can be enormous. For example, the sequence *ababaabb* is covered by the following DTDs in addition to many more – $(a \mid b)^*$, $(a \mid b)^*a^*b^*$, $(ab)^*(a \mid b)^*$, $(ab)^*a^*b^*$. Therefore, in this paper, we outline several novel heuristics, inspired by real-life DTDs[2], for limiting the set of candidate DTDs $\mathcal{S}_\mathcal{G}$ output by the generalization module.

The Factoring Subsystem. The factoring component factors two or more candidate DTDs in $\mathcal{S}_\mathcal{G}$ into a new candidate DTD. The length of the new DTD is smaller than the sum of the sizes of the DTDs factored. For example, in Figure 3(a), candidate DTDs b^*d and b^*e representing the expression $b^*d \mid b^*e$, when factored, result in the DTD $b^*(d \mid e)$; similarly, the candidates *ac*, *ad*, *bc* and *bd* are factored into $(a \mid b)(c \mid d)$ (the pre-factored expression is $ac \mid ad \mid bc \mid bd$). Although factoring leaves the semantics of candidate DTDs unchanged, it is nevertheless an important step. The reason being that factoring reduces the size of the DTD and thus the cost of encoding the DTD, without seriously impacting the cost of encoding input sequences using the DTD. Thus, since the DTD encoding cost is a component of the MDL cost for a DTD, factoring can result in certain DTDs being chosen by the MDL module that may not have been considered earlier. We appropriately modify factoring algorithms for boolean functions in the logic optimization area [BM82, Wan89] to meet our needs. However, even though every subset of candidate DTDs can, in principle, be factored, the number of these subsets can be large and only a few of them result in good factorizations. We propose novel heuristics to restrict our attention to subsets that can be factored effectively.

The MDL Subsystem. The MDL subsystem finally chooses from among the set of candidate DTDs $\mathcal{S}_\mathcal{F}$ generated by the previous two subsystems, a set of DTDs that cover all the input sequences in I and the sum of whose MDL costs is minimum. The final DTD is then an *or* of the DTDs in the set. For the input sequences in Figure 3(a), we illustrate (using solid lines) in Figure 3(b), the input sequences (in the right column) covered by the candidate DTDs in $\mathcal{S}_\mathcal{F}$ (in the left column).

The above cost minimization problem naturally maps to the *Facility Location Problem* (FLP) for which polynomial time approximation algorithms have been proposed in the literature [Hoc82, CG99]. We adapt the algorithm from [CG99] for our purposes, and using it, the XTRACT system is able to infer the DTD shown at the bottom of Figure 3(b).

4 The MDL Subsystem

The MDL subsystem constitutes the core of the XTRACT system – it is responsible for choosing a set $\mathcal{S}$ of candidate DTDs from $\mathcal{S}_\mathcal{F}$ such that the final DTD $\mathcal{D}$ (which is an *or* of the DTDs in $\mathcal{S}$) (1) covers all sequences in I, and (2) has the minimum MDL cost. Consequently, we describe this module first, and postpone the presentation of the generalization and factoring modules to Sections 5 and 6, respectively.

Recall that the MDL cost of a DTD that is used to explain a set of sequences, comprises of

(A) the length, in bits, needed to describe the DTD, and

(B) the length of the sequences, in bits, when encoded in terms of the DTD.

Thus, in the following subsection, we first present the encoding schemes for computing parts (A) and (B) of the MDL cost of a DTD. Subsequently, in Section 4.2, we present the algorithm for computing the set $\mathcal{S} \subseteq \mathcal{S}_\mathcal{F}$ of candidate DTDs whose *or*

[2]The DTDs are available at Robin Cover's SGML/XML Web page (http://www.oasis-open.org/cover/).

(A) $seq(D, s) = \epsilon$ if $D = s$. In this case, the DTD D is a sequence of symbols from the alphabet Σ and does not contain any metacharacters.

(B) $seq(D_1 \ldots D_k, s_1 \ldots s_k) = seq(D_1, s_1) \ldots seq(D_k, s_k)$ that is, D is the concatenation of regular expressions $D_1 \ldots D_k$ and the sequence s can be written as the concatenation of the subsequences $s_1 \ldots s_k$, such that each subsequence s_i matches the corresponding regular expression D_i.

(C) $seq(D_1 | \ldots | D_m, s) = i \; seq(D_i, s)$ that is, D is the exclusive choice of regular expressions $D_1 \ldots D_m$, and i is the index of the regular expression that the sequence s matches. Note that we need $\lceil \log m \rceil$ bits to encode the index i.

(D) $seq(D^*, s_1 \ldots s_k) = \begin{cases} k \; seq(D, s_1) \ldots seq(D, s_k) & \text{if } k > 0 \\ 0 & \text{otherwise} \end{cases}$
In other words, the sequence $s = s_1 \ldots s_k$ is produced from D^* by instantiating the repetition operator k times, and each subsequence s_i matches the i-th instantiation. In this case, since there is no simple and inexpensive way to bound apriori, the number of bits required for the index k, we first specify the number of bits required to encode k in unary (that is, a sequence of $\lceil \log k \rceil$ 1s, followed by a 0) and then the index k using $\lceil \log k \rceil$ bits. The 0 in the middle serves as the delimiter between the unary encoding of the length of the index and the actual index itself.

Figure 4: The Encoding Scheme

yields the final DTD $\mathcal{D}$ with the minimum MDL cost. Note that the candidate DTDs in $\mathcal{S}_{\mathcal{F}}$ can be complex regular expressions (containing *, $|$ etc.) output by the generalization and factoring subsystems.

4.1 The encoding scheme

We begin by describing the procedure for estimating the number of bits required to encode the DTD itself (part (A) of the MDL cost). Let Σ be the set of subelement symbols that appear in sequences in I. Let $\mathcal{M}$ be the set of metacharacters $|, ^*, +, ?, (,)$. Let the length of a DTD viewed as a string in $\Sigma \cup \mathcal{M}$, be n. Then, the length of the DTD in bits is $n \log(|\Sigma| + |\mathcal{M}|)$. As an example, let Σ consist of the elements a and b. The length in bits of the DTD a^*b^* is $4 * \log(2 + 6) = 12$. Similarly, the length in bits of the DTD $(ab|abb)(aa|ab^*)$ is $16 * 3 = 48$.

We next describe the scheme for encoding a sequence using a DTD (part (B) of the MDL cost). The encoding scheme constructs a sequence of integral indices (which forms the encoding) for expressing a sequence in terms of a DTD. The following simple examples illustrate the basic building blocks on which our encoding scheme for more complex DTDs is built:

1. The encoding for the sequence a in terms of the DTD a is the empty string ϵ.

2. The encoding for the sequence b in terms of the DTD $a \mid b \mid c$ is the integral index 1 (denotes that b is at position 1, counting from 0, in the above DTD).

3. The encoding for the sequence bbb in terms of the DTD b^* is the integral index 3 (denotes 3 repetitions of b).

We now generalize the encoding scheme for arbitrary DTDs and arbitrary sequences. Let us denote the sequence of integral indices for a sequence s when encoded in terms of a DTD D by $seq(D, s)$. We define $seq(D, s)$ recursively in terms of component DTDs within D as shown in Figure 4. Thus, $seq(D, s)$ can be computed using a recursive procedure based on the encoding scheme of Figure 4. Note that we have not provided the definitions of the encodings for operators $^+$ and ? since these can be

defined in a similar fashion to * (for $^+$, k is always greater than 0, while for ?, k can only assume values 1 or 0). We now illustrate the encoding scheme using the following example.

Example 4.1 Consider the DTD $(ab|c)^*(de|fg^*)$ and the sequence $abccabfggg$ to be encoded in terms of the DTD. Below, we list how steps (A), (B), (C) and (D) in Figure 4 are recursively applied to derive the encoding $seq((ab|c)^*(de|fg^*), abccabfggg)$.

1. **Apply Step (B).** $seq((ab|c)^*, abccab))seq((de|fg^*), fggg)$
2. **Apply Step (D).** 4 $seq(ab|c, ab)$ $seq(ab|c, c)$ $seq(ab|c, c)$ $seq(ab|c, ab)$ $seq((de|fg^*), fggg)$
3. **Apply Step (C).** 4 0 $seq(ab, ab)$ 1 $seq(c, c)$ 1 $seq(c, c)$ 0 $seq(ab, ab)$ 1 $seq(fg^*, fggg)$
4. **Apply Step (A).** 4 0 1 1 0 1 $seq(fg^*, fggg)$
5. **Apply Steps (A), (B) and (D).** 4 0 1 1 0 1 3

In order to derive the final bit sequence corresponding to the above indices, we need to include in the encoding the unary representation for the number of bits required to encode the indices 4 and 3. Thus, we obtain the following bit encoding for the sequence (we have inserted blanks in between the encoding for successive indices for clarity).

$$seq((ab|c)^*(de|fg^*), abccabfggg) = 1110100\ \ 0\ \ 1\ \ 1\ \ 0\ \ 1\ \ 11011$$

□

In steps (B), (C) and (D), we need to be able to determine if a sequence s matches a DTD D. Since a DTD is a regular expression, well-established techniques for finding out if a sequence is covered by a regular expression can be used for this purpose [HU79] and have a complexity of $O(|D| \cdot |s|)$ ($|s|$ denotes the length of sequence s). These methods involve constructing a non-deterministic finite automaton for D and can also be used to decompose the sequence s into subsequences such that each subsequence matches the corresponding sub-part of the DTD D, thus enabling us to come up with the encoding.

Note that there may be multiple ways of partitioning the sequence s such that each subsequence matches the corresponding sub-part of the DTD D. In such a case, we can extend the above procedure to enumerate every decomposition of s that match sub-parts of D, and then select from among the decompositions, the one that results in the minimum length encoding of s in terms of D. The complexity of considering all possible decompositions, however, is much higher and therefore not included in our XTRACT implementation.

4.2 Computing the DTD with Minimum MDL Cost

We now turn our attention to the problem of computing the final DTD $\mathcal{D}$ (which is an *or* of a subset $\mathcal{S}$ of candidate DTDs in $\mathcal{S}_{\mathcal{F}}$) that covers all the input sequences in I and whose MDL cost for encoding sequences in I is minimum. The above minimization problem maps naturally to the *Facility Location Problem* (FLP) [Hoc82, CG99]. The FLP is formulated as follows: Let C be a set of clients and J be a set of facilities such that each facility "serves" every client. There is a cost $c(j)$ of "choosing" a facility $j \in J$ and a cost $d(j, i)$ of serving client $i \in C$ by facility $j \in J$. The problem definition asks to choose a subset of facilities $F \subset J$ such that the sum of costs of the chosen facilities plus the sum of costs of serving every client by its closest chosen facility is minimized, that is

$$\min_{F \subset J}\{\sum_{j \in F} c(j) + \sum_{i \in C} \min_{j \in F} d(j, i)\} \tag{1}$$

The problem of inferring the minimum MDL cost DTD can be reduced to FLP as follows: Let C be the set I of input sequences and J be the set of candidate DTDs in $\mathcal{S}_{\mathcal{F}}$. The cost of choosing a facility is the length of the corresponding candidate DTD. The cost of serving client i from facility j, $d(j, i)$, is the length of the encoding of the sequence corresponding to i using the DTD corresponding to the facility j. If a DTD j does not cover a sequence i, then we set $d(j, i)$ to ∞. Thus, the set F computed by the FLP corresponds to our desired set $\mathcal{S}$ of candidate DTDs.

The FLP is NP-hard; however, it can be reduced to the *set cover problem* and then approximated within a logarithmic factor as shown in [Hoc82]. In our implementation, we employ the randomized algorithm from [CG99] which approximates the FLP within a constant factor if the distance function is a metric. Even though our distance function is not a metric, we have found the FLP approximations produced by [CG99] for our problem setting to be very good in practice. Furthermore, the time complexity of [CG99] for computing the approximate solution is $O(N^2 \cdot \log N)$, where $N = |I|$.

5 The Generalization Subsystem

The quality of the DTD computed by the MDL module is very dependent on the set of candidate DTDs $\mathcal{S}_{\mathcal{F}}$ input to it. In case $\mathcal{S}_{\mathcal{F}}$ were to contain only input sequences in I, then the final DTD output by the MDL subsystem would simply be the *or* of all the sequences in I. However, as we observed earlier, this is not a desirable DTD since it is neither concise nor intuitive. Thus, in order to infer meaningful DTDs, it is crucial that the candidate DTDs in $\mathcal{S}_{\mathcal{F}}$ be general – the goal of the generalization component is to achieve this objective by inferring a set $\mathcal{S}_{\mathcal{G}}$ of general DTDs which are then input to the factorization step. As we mentioned before, the factorization step infers additional factored DTDs and generates $\mathcal{S}_{\mathcal{F}}$ which is a superset of $\mathcal{S}_{\mathcal{G}}$.

The generalization component in XTRACT infers a number of regular expressions which we have found to frequently appear in real-life DTDs. Below, we present examples of such regular expressions from real-life DTDs that appear in the Newspaper Association of America (NAA) Classified Advertizing Standards XML DTD[3].

a^*bc^*: DTDs of this form are generally used to specify tuples with set-valued attributes.

```
<!ELEMENT account-info (account-number, sub-account-number*)> <!--
Specification for account identification information -->
```

$(abc)^*$: This type of DTD is used to represent a set (or a list) of ordered tuples.

```
<!ELEMENT days-and-hours (date, time)+> <!-- provide times/dates
when job fairs will be held -->
```

$(a|b|c)^*$: The DTD of the form $(a|b|c)*$ is frequently used to represent a multiset containing the elements a, b and c. This DTD is very useful since the elements in the multiset are allowed to appear multiple times and in any order in the document. For example, the following DTD specifies that the support information for an ad can consist of an arbitrary number of audio or video clips, photos, and further these can appear in any order.

```
<!ELEMENT support-info (audio-clip | file-id | graphic | logo |
new-list | photo | video-clip | zz-generic-tag)*> <!-- support
information for ad content -->
```

$((ab)^*c)^*$: This type of DTD permits nesting relationships among sets (or lists).

```
<!ELEMENT transfer-info (transfer-number, (from-to, company-id)+,
contact-info)*> <!-- provides parent information through the multi-
level aggregation process. may be repeated -->
```

Although our algorithms can infer regular expressions that are more complex than the above, we do not infer certain complex expressions such as $(ab?c^*d?)*$ that are less likely to occur in practice. We defer further discussion of this topic to Section 7.

We now discuss our generalization algorithm which is outlined in Figure 5. Procedure GENERALIZE infers several DTDs for each input sequence in I independently and adds them to the set $\mathcal{S}_{\mathcal{G}}$. Therefore, it may over-generalize in some cases (since we

[3]These can be accessed at http://www.naa.org/technology/clsstdtf/Adex010.dtd.

are inferring DTDs based on a single sequence), but however, our MDL step will ensure that such over-general DTDs are not chosen as part of the final inferred DTD, if there are better alternatives. Recall that the generalization step is merely trying to provide several alternate candidates to the MDL step. In particular, $S_{\mathcal{G}} \supseteq I$, and therefore, the DTD corresponding to the *or*'s of the input will be considered by the MDL step.

The essence of procedure GENERALIZE are the procedures DISCOVERSEQPATTERN and DISCOVERORPATTERN which are repeatedly called with various parameter values. We discuss details of these procedures and the roles of the parameters next.

5.1 Discovering Sequencing Patterns

Procedure DISCOVERSEQPATTERN, shown in Figure 5, takes as input an input sequence s and returns a candidate DTD that is derived from s by replacing sequencing patterns of the form $xx\cdots x$, for a subsequence x in s, with the regular expression $(x)^*$. In addition to s, the procedure also accepts as input, a threshold parameter $r > 1$ which is the minimum number of contiguous repetitions of subsequence x in s required for the repetitions to be replaced with $(x)^*$. In case there are multiple subsequences x with the maximum number of repetitions in Step 2, the longest among them is chosen, and subsequent ties are resolved arbitrarily.

Note that instead of introducing the regular expression term $(x)^*$ into the sequence s, we choose to introduce an *auxiliary* symbol that serves as a representative for the term. The auxiliary symbols enable us to keep the description of our algorithms simple and clean since the input to them is always a sequence of symbols. We ensure that there is a one-to-one correspondence between auxiliary symbols and regular expression terms throughout the XTRACT system; thus, if the auxilliary symbol, A denotes $(bc)^*$ in one candidate DTD, then it represents $(bc)^*$ in every other candidate DTD. Also observe that procedure DISCOVERSEQPATTERN may perform several iterations and thus new sequencing patterns may contain auxiliary symbols corresponding to patterns replaced in previous iterations. For example, invoking procedure DISCOVERSEQPATTERN with the input sequence $s = abababcababc$ and $r = 2$ yields the sequence A_1cA_1c after the first iteration, where A_1 is an auxiliary symbol for the term $(ab)^*$. After the second iteration, the procedure returns the candidate DTD A_2, where A_2 is the auxiliary symbol corresponding to $((ab)^*c)^*$. Thus, the resulting candidate DTD returned by procedure DISCOVERSEQPATTERN can contain *s nested within other *s. Finally, we have chosen to invoke DISCOVERSEQPATTERN (from GENERALIZE) with three different values for the parameter r to control the eagerness with which we generalize. For example, for the sequence $aabbb$, DISCOVERSEQPATTERN with $r = 2$ would infer a^*b^*, while with $r = 3$, it would infer aab^*. In the MDL step, if many other sequences are covered by aab^*, then a DTD of aab^* may be preferred to a DTD of a^*b^* since it more accurately describes sequences in I.

The time complexity of the procedure is dominated by the first step that involves finding the subsequence x with the maximum number of contiguous repetitions. Since s contains at most $O(|s|^2)$ possible subsequences and computing the number of repetitions for each subsequence takes $O(|s|)$ steps, the complexity of the first step is $O(|s|^3)$ per iteration, in the worst case.

5.2 Discovering Or Patterns

Procedure DISCOVERORPATTERN infers patterns of the form $(a_1|a_2|\ldots|a_m)^*$ based on the locality of these symbols within a sequence s. It finds out such locality by first partitioning (performed by procedure PARTITION) the input sequence s into the smallest possible subsequences $s_1, s_2, \ldots, s_n$, such that for any occurrence of a symbol a in a subsequence s_i, there does not exist another occurrence of a in some other subsequence s_j within a distance d (which is a parameter to DISCOVERORPATTERN). Each subsequence s_i in s is then replaced by the pattern $(a_1|a_2|\ldots|a_m)^*$ where $a_1, \ldots, a_m$ are the distinct symbols in the subsequence s_i. The intuition here is that if s_i contains frequent repetitions of the symbols $a_1, a_2, \ldots, a_m$ in close proximity, then it is very likely that s_i originated from a regular expression of the form $(a_1|a_2|\ldots|a_m)^*$. As an illustration, on the input sequence $abcbac$, procedure DISCOVERORPATTERN returns

- aA_1ac for $d = 2$, where $A_1 = (b \mid c)^*$,
- aA_2 for $d = 3$, where $A_2 = (a \mid b \mid c)^*$, and
- A_2 for $d = 4$, where $A_2 = (a \mid b \mid c)^*$.

```
procedure GENERALIZE(I)
begin
    for each sequence s in I
        add s to S_G
        for r := 2, 3, 4
            s' := DISCOVERSEQPATTERN(s, r)
            for d := 0.1 · |s'|, 0.5 · |s'|, |s'|
                s'' := DISCOVERORPATTERN(s', d)
                add s'' to S_G
end

procedure DISCOVERSEQPATTERN(s, r)
begin
    repeat
        let x be a subsequence of s with the maximum number (≥ r) of contiguous repetitions in s
        replace all (≥ r) contiguous occurrences of x in s with a new auxiliary symbol A_i = (x)*
    until (s no longer contains ≥ r contiguous occurrences of any subsequence x)
    return s
end

procedure DISCOVERORPATTERN(s, d)
begin
    s_1, s_2, ..., s_n := PARTITION(s, d)
        for each subsequence s_j in s_1, s_2, ..., s_n
            let the set of distinct symbols in s_j be a_1, a_2, ..., a_m
            if (m > 1)
                replace subsequence s_j in sequence s by a new auxiliary symbol A_i = (a_1 | ··· | a_m)*
    return s
end

procedure PARTITION(s, d)
begin
    i := start := end := 1
    s_i = s[start, end]
    while (end < |s|)
        while (end < |s| and a symbol in s_i occurs to the right of s_i within a distance d)
            end := end + 1; s_i := s[start, end]
        if (end < |s|)
            i := i + 1; start := end + 1; end := end + 1; s_i := s[start, end]
    return s_1, s_2, ···, s_i
end
```

Figure 5: The Generalization Algorithm

A critical component for discovering or patterns is procedure PARTITION, which we now discuss in more detail. Before that, we define the following notation for sequences. For a sequence s, $s[i,j]$ denotes the subsequence of s starting at the i^{th} symbol and ending at the j^th symbol of s. Procedure PARTITION constructs the subsequences in the order s_1, s_2, and so on. Assuming that s_1 through s_j have been generated, it constructs s_{j+1} by starting s_{j+1} immediately after s_j ends and expanding the subsequence s_{j+1} to the right as long as required to ensure that there is no symbol in s_{j+1} that occurs within a distance d to the right of s_{j+1}. By construction, there cannot exist such a symbol to the left of s_{j+1}. Note that the condition whether a symbol in s_i occurs within a distance d outside s_i can be checked in $O(|s|)$ time if we keep track of the next occurrence outside s_i of every symbol in s_i – this can be achieved by initially constructing for every symbol, the locations of its occurrences in s sorted order. Therefore, the time complexity of procedures PARTITION and DISCOVERORPATTERN can be easily shown to be $O(|s|^2)$.

Note that procedure GENERALIZE invokes DISCOVERORPATTERN on the DTDs that result from calls to DISCOVERSEQPATTERN and therefore it is possible to infer more complex DTDs of the form $(a|(bc)^*)^*$ in addition to DTDs like $(a|b|c)^*$. For instance, for the input sequence $s = abcbca$, procedure DISCOVERSEQPATTERN invoked with $r = 2$ would return $s' = aA_1a$, where $A_1 = (bc)^*$, which when input to DISCOVERORPATTERN returns $s'' = A_2$ for $d = |s'|$, where $A_2 = (a|A_1)^*$. Further, observe that DISCOVERORPATTERN is invoked with various values of d (expressed as a fraction of the length of the input sequence) to control the degree of generalization. Small values of d lead to conservative generalizations while larger values result in more liberal generalizations.

6 The Factoring Subsystem

In a nutshell, the factoring step derives factored forms for expressions that are an *or* of a subset of the candidate DTDs in $\mathcal{S_G}$. For example, for candidate DTDs *ac*, *ad*, *bc* and *bd* in $\mathcal{S_G}$, the factoring step would generate the factored form $(a \mid b)(c \mid d)$. Note that since the final DTD is an *or* of candidate DTDs in $\mathcal{S_F}$, factored forms are candidates, too. Further, a factored candidate DTD, because of its smaller size, has a lower MDL cost, and is thus more likely to be chosen in the MDL step. Thus, since factored forms (due to their compactness) are more desirable (see restriction R1 in Section 3), factoring can result in better quality DTDs. In this section, we describe the algorithms used by the factoring module to derive factored forms of the candidate DTDs in $\mathcal{S_G}$ produced by the generalization step.

Factored DTDs are common in real life, when there are several choices to be made. For example, in the DTD in Figure 2, an article may be categorized based on whether it appeared in a workshop, conference or journal; it may also be classified according to its area as belonging to either computer science, physics, chemistry etc. Thus, the DTD (in factored form) for the element `article` would then be as follows:

```
<!ELEMENT article(title, author*, (workshop | conference | journal),
(computer science | physics | chemistry | ...))
```

The set of candidate DTDs output by the factorization module, $\mathcal{S_F}$, in addition to the factored forms generated from candidates in $\mathcal{S_G}$, also contains all the DTDs in $\mathcal{S_G}$. Ideally, factored forms for every subset of $\mathcal{S_G}$, should be added to $\mathcal{S_F}$ to be considered by the MDL module. However, this is clearly impractical, since $\mathcal{S_G}$ could be pretty large. Therefore, in the following subsection, we propose a heuristic for selecting sets of candidates in $\mathcal{S_G}$ that when factored yield good factored DTDs. We then present a brief description of the factoring algorithm itself, which is an adaptation of factoring algorithms for boolean expressions from the logic optimization literature.

Note that each candidate DTD in $\mathcal{S_G}$ is a sequence of symbols, some of which can be auxiliary symbols. Recall that auxiliary symbols translate to regular expressions on symbols in Σ, and there is a one-to-one correspondence between auxiliary symbols and the expressions that they represent.

```
procedure FACTORSUBSETS(S_G)
begin
for each DTD D in S_G
        Compute score(D,S_G)
S_F := S' := S_G; SeedSet := ∅
for i := 1 to k
        let D be the DTD in S' with the maximum value for score(D,S_G)
        SeedSet := SeedSet ∪ D
        S' := S' − {D' : overlap(D, D') ≥ δ}
for each DTD D in SeedSet
        S := {D}
        S' := S_G − {D' : overlap(D, D') ≥ δ}
        while (S' is not empty)
                let D' be the DTD in S' with the maximum value for score(D', S)
                S := S ∪ D'
                S' := S' − {D'' : overlap(D', D'') ≥ δ}
        F := FACTOR(S)
        S_F := S_F ∪ {F_1, ..., F_m}        /* F = F_1 | ··· | F_m */
end
```

Figure 6: Choosing Subsets Of $\mathcal{S}_\mathcal{G}$ For Factoring

6.1 Selecting Subsets of $\mathcal{S}_\mathcal{G}$ to Factor

In this section, we describe how we choose subsets of $\mathcal{S}_\mathcal{G}$ that lead to good factorizations. Intuitively, a subset S of $\mathcal{S}_\mathcal{G}$ is a good candidate for factoring if the factored form of S is much smaller than S itself. In addition, even though $\mathcal{S}_\mathcal{G}$ may contain multiple generalizations that are derived from the same input sequence, it is highly unlikely that the final DTD will contain two generalizations of the same input sequence. Thus, factoring candidate DTDs in $\mathcal{S}_\mathcal{G}$ that cover similar sets of input sequences does not lead to factors that can improve the quality of the final DTD.

We thus conclude that if a subset S of $\mathcal{S}_\mathcal{G}$ to yield good factored forms it must satisfy the following two properties:

1. Every DTD in S has a common prefix or suffix with a number of other DTDs in S. Further, as more DTDs in S share common prefixes or suffixes, or as the length of the common prefixes/suffixes increases, the quality of the generated factored form can be expected to improve.

2. The *overlap* between every pair of DTDs D, D' in S is minimal, that is, the intersection of the input sequences covered by D and D' is small. This is important because, as mentioned above, a factored DTD adds little value (from an MDL cost perspective) over the candidate DTDs from which it was derived if it cannot be used to encode a significantly larger number of input sequences compared to the sequences covered by each individual DTD.

Definitions. In order to state properties (1) and (2) for a set S of DTDs more formally, we need to first define the following notation. For a DTD D, let $cover(D)$ denote the input sequences in I that are covered by D (note that auxiliary symbols are expanded completely when *cover* for a DTD is computed). Then, $overlap(D, D')$ is defined as the fraction of the input sequences covered by D and D' that are common to D and D', that is, $overlap(D, D') = \frac{|cover(D) \cap cover(D')|}{|cover(D) \cup cover(D')|}$. Thus, for a sufficiently

small value of the (user-specified) parameter δ, by ensuring that $overlap(D, D') < \delta$ for every pair of DTDs D and D' in S, we can ensure that S satisfies Property (2) mentioned above.

In order to characterize Property (1) more rigorously, we introduce the function $score(D, S)$ which attempts to capture the degree of similarity between prefixes/suffixes of DTD D and those of DTDs in the set S of DTDs. Intuitively, a DTD with a high *score* with respect to set S is a good candidate to be factored with other DTDs in set S. For a DTD D, let $pref(D)$ and $suf(D)$ denote the set of prefixes and suffixes of D, respectively. Let $psup(p, S)$ denote the support of prefix p in set S of DTDs, that is, the number of DTDs in S for which p is a prefix. Similarly, let $ssup(s, S)$ denote number of DTDs in S for which s is a suffix. Then $score(D, S)$ is defined as follows.

$$score(D,S) = \max(\{|p| \cdot psup(p,S) : p \in pref(D)\} \cup \{|s| * ssup(s,S) : s \in suf(D)\})$$

Thus, the prefix/suffix p/s of D, for which the product of p/s's length and its support in S is maximum, determines the score of D with respect to S. The intuition here is that if DTD D has a long prefix or suffix that occurs frequently in set S, then this prefix can be factored out thus resulting in good factored forms. The function *score* is thus a good measure of how well D would factor with other DTDs in S.

Algorithm. Procedure FACTORSUBSETS, shown in Figure 6, first selects subsets S of S_G to factor that satisfy properties (1) and (2) mentioned earlier. Each of these subsets S is then factored by invoking procedure FACTOR (in Step 15) described in the next subsection. Assuming that the factoring algorithm returns $F_1 \mid F_2 \mid \cdots F_m$, each of the F_i is added to $S_{\mathcal{F}}$ that is then input to the MDL module.

We now discuss how procedure FACTORSUBSETS computes the set S of candidate DTDs to factor. First, k seed DTDs for the sets S to be factored are chosen in the for loop spanning steps 4–7. These seed DTDs have a high *score* value with respect to S_G and overlap minimally with each other. Thus, we ensure that each seed DTD not only factors well with other DTDs in S_G, but is also is significantly different from other seeds. In steps 9–14, each seed DTD is used to construct a new set S of DTDs to be factored (thus, only k sets of DTDs are generated). After initializing S to a seed DTD D, in each subsequent iteration, the next DTD D' that is added to S is chosen greedily – it is the one whose score with respect to DTDs in S is maximum and whose overlap with DTDs already in S is less than δ.

Complexity Results. The time complexity of selecting the sets S to factor in the FACTORSUBSETS procedure can be shown to be $O(N^2 \cdot (N + L))$, where $N = |I|$ and L is the maximum length of an input sequence in I. The reason for this is that the initial computation of $score(D, S_G)$ for every DTD D in S_G requires us to compute the support of every prefix and suffix of D in S_G. Since S_G contains $O(N)$ DTDs, and each DTD can have at most $2L$ prefixes/suffixes, there are at most $O(N \cdot L)$ distinct prefixes and suffixes. The supports for these can be computed in $O(N \cdot L)$ steps by storing them in a trie structure. Thus, the time complexity of computing the scores for all the DTDs in S_G (in steps 1–2) is $O(N \cdot L)$.

Computing the overlap between a pair of DTDs requires $O(N)$ time to compute the intersection and union of the input sequences they cover. Thus, the worst-case time complexity to compute the overlap between all pairs of DTDs in S_G is $O(N^3)$. Assuming that we precompute the overlapping DTD pairs in S_G, SeedSet can be computed in $O(N)$ steps (since the number of seeds, k, is a constant). Furthermore, the time complexity of computing each set S of DTDs to be factored can be shown to be $O(N^2 \cdot L)$ since the while loop (steps 11–14) performs at most $O(N)$ iterations and the cost of recomputing the scores for DTDs in S' (with respect to S) in each iteration is $O(N \cdot L)$ (as before, this can be achieved by maintaining a trie structure for prefixes and suffixes of DTDs in S).

6.2 Algorithm For Factoring a Set of DTDs

In this section, we show how the factored form for a set S of DTDs can be derived – the expression we factor is actually the *or* of the DTDs in S. Algorithms for computing the optimum factored form, that is, the one with the minimum number of literals have been proposed earlier in [Law64]. However, the complexity of these exact techniques are impractical for all but the smallest

7.2 Data Sets

In order to evaluate the quality of DTDs retrieved by XTRACT, we used both synthetic as well as real-life DTD schemas. For each DTD for a single element, we generated an XML file containing 1000 instantiations of the element. These 1000 instantiations were generated by randomly sampling from the DTD for the element. Thus, the initial set of input sequences I to both XTRACT and DDbE contained somewhere between 500 and 1000 sequences (after the elimination of duplicates) conforming to the original DTD.

Synthetic DTD Data Set. We used a synthetic data generator to generate the synthetic data sets. Each DTD is randomly chosen to have one of the following two forms: $A_1|A_2|A_3|\cdots|A_n$ and $A_1A_2A_3\cdots A_n$. Thus, a DTD has n building blocks where n is randomly chosen number between 1 and mb, where mb is an input parameter to the generator that specifies the maximum number of building blocks in a DTD. Each building block A_i further consists of n_i symbols, where n_i is randomly chosen to be between 1 and ms (the parameter ms specifies the maximum number of symbols that can be contained in a building block). Each building block A_i has one of the following four forms, each of which has an equal probability of occurrence: (1) $(a_1|a_2|a_3|\ldots|a_{n_i})$ (2) $a_1a_2a_3\ldots a_{n_i}$ (3) $(a_1|a_2|a_3|a_4|\ldots|a_{n_i})^*$ (4) $(a_1a_2a_3a_4\ldots a_{n_i})^*$. Here, the a_i's denote subelement symbols. Thus, our synthetic data generator essentially generates DTDs containing one level of nesting of regular expression terms.

In Table 3, we show the synthetic DTDs that we considered in our experiments (note that, in the figure, we only include the regular expression corresponding to the DTD). The DTDs were produced using our generator with the input parameters mb and ms both set to 5. Note that we use letters from the alphabet as subelement symbols.

No.	Original DTD
1	$abcde\|efgh\|ij\|klm$
2	$(a\|b\|c\|d\|f)^*gh$
3	$(a\|b\|c\|d)^*\|e$
4	$(abcde)^*f$
5	$(ab)^*\|cdef\|(ghi)^*$
6	$abcdef(g\|h\|i\|j)(k\|l\|m\|n\|o)$
7	$(a\|b\|c)d^*e^*(fgh)^*$
8	$(a\|b)(cdefg)^*hijklmnopq(r\|s)^*$
9	$(abcd)^*\|(e\|f\|g)^*\|h\|(ijklm)^*$
10	$a^*\|(b\|c\|d\|e\|f)^*\|gh\|(i\|j\|k)^*\|(lmn)^*$

Table 3: Synthetic DTD Data Set

The ten synthetic DTDs vary in complexity with later DTDs being more complex than the earlier ones. For instance, DTD 1 does not contain any metacharacters, while DTDs 2 through 5 contain simple sequencing and or patterns. DTD 6 represents a DTD in factored form while in DTDs 7 through 10, factors are combined with sequencing and or patterns.

Real-life DTD Data Set. We obtained our real-life DTDs from the Newspaper Association of America (NAA) Classified Advertising Standards XML DTD produced by the NAA Classified Advertising Standards Task Force[5]. We examined this real-life DTD data and collected six representative DTDs that are shown in Table5. Of the DTDs shown in the table, the last three DTDs are quite interesting. DTD 4 contains the metacharacter ? in conjunction with the metacharacter *, while DTDs 5 and 6 contain two regular expressions with *'s, one nested within the other.

<table>
<tr><th>No.</th><th>Original DTD</th><th>Simplified DTD</th></tr>
<tr><td>1</td><td>< !ENTITY % included-elements
"audio-clip|blind-box-reply|graphic|linkpi-char|video-clip" ></td><td>$a|b|c|d|e$</td></tr>
<tr><td>2</td><td>< !ELEMENT communications-contacts
(phone|fax|email|pager|web-page)* ></td><td>$(a|b|c|d|e)^*$</td></tr>
<tr><td>3</td><td>< !ELEMENT employment-services(employment-service.type,
employment-service.location * (e.zz-generic-tag)*) ></td><td>ab^*c^*</td></tr>
<tr><td>4</td><td>< !ENTITY % location"addr*, geographic-area?, city?, state-province?,
postal-code?, country?" ></td><td>$a^*b?c?d?$</td></tr>
<tr><td>5</td><td>< !ELEMENT transfer-info(transfer-number, (from-to, company-id)+,
contact-info)* ></td><td>$(a(bc)^+d)^*$</td></tr>
<tr><td>6</td><td>< !ELEMENT real-estate-services(real-estate-service.type,
real-estate-service.location?, r-e.response-modes*, r-e.comment?)* ></td><td>$(ab?c^*d?)^*$</td></tr>
</table>

Table 4: Real-life DTD Data Set

7.3 Quality of Inferred DTDs

Synthetic DTD Data Set. The DTDs inferred by XTRACT and DDbE for the synthetic data set are presented in Table 3. As shown in the table, XTRACT infers each of the original DTDs correctly. In contrast, DDbE computes the accurate DTD for only DTD 1 which is the simplest DTD containing no metacharacters. Even for the simple DTDs 2–5, not only is DDbE unable to correctly deduce the original DTD, but it also infers a DTD that does not cover the set of input sequences. For instance, one of the input sequences covered by DTD 2 is gh which is not covered by the DTD inferred by DDbE. Thus, while XTRACT infers a DTD that covers all the input sequences, the DTD returned by DDbE may not cover every input sequence. DTD 4 exemplifies the two typical behaviors of DDbE – (1) sequence f that is not frequently repeated is appended to both the front and the back of the final DTD, and (2) symbols that are repeated frequently are all *or*'d together and encapsulated by the metacharacter $^+$. For example, DDbE incorrectly identifies the term $(abcde)^*$ to be $(a|b|c|d|e)^*$ which is much more general. Thus, the DDbE tool has a tendency to over-generalize when the original DTDs contain regular expressions with *s. This same trend to over-generalize can be seen in DTDs 8–10 also. On the other hand, as is evident from Table 3, this is not the case for XTRACT which correctly infers every one of the original DTDs even for the more complex DTDs 8–10 that contain various combinations of sequencing and or patterns. This clearly demonstrates the effectiveness of our generalization module in discovering these patterns and our MDL module in selecting these general candidate DTDs as the final DTDs.

Also, as discussed earlier, DDbE is not very good at factoring DTDs. For instance, unlike XTRACT, DDbE is unable to derive the final factored form for DTD 6. Finally, DDbE infers an extremely complex DTD for the simple DTD 7. The results for the synthetic data set clearly demonstrate the superiority of XTRACT's approach (based on the combination of generalization, factoring and the MDL principle) compared to DDbE's for the problem of inferring DTDs.

Real-life DTD Data Set. The DTDs generated by the two algorithms for the real-life data set are shown in Table 6. Of the six DTDs, XTRACT is able to infer the first five correctly. In contrast, DDbE is able to derive the accurate DTD only for DTDs 1 and 2, and an approximate DTD for DTD 3. Basically, with an additional factoring step, DDbE could obtain the original DTD for DTD 3. Note, however, that DDbE is unable to infer the simple DTD 4 that contains the metacharacter ?. In contrast, XTRACT is able to deduce this DTD because it's factorization step takes into account the identity element "1" and simplifies expressions of the form $1|a$ to $a?$. DTD 5 represents an interesting case where XTRACT is able to mine a DTD containing regular expressions containing nested *s. This is due to our generalization module that iteratively looks for sequencing patterns. On the other hand, DDbE simply over-generalizes DTD 5 by *or*ing all the symbols in it and enclosing them within the metacharacter $^+$. Finally,

[5] This can be accessed at http://www.naa.org/technology/clsstdtf/Adex010.dtd

No.	Original DTD	DTD Inferred by XTRACT	DTD Inferred by DDbE
1	$abcde \mid efgh \mid ij \mid klm$	$abcde \mid efgh \mid ij \mid klm$	$abcde \mid efgh \mid ij \mid klm$
2	$(a \mid b \mid c \mid d \mid f)^* gh$	$(a \mid b \mid c \mid d \mid f)^* gh$	$gh(a \mid b \mid c \mid d \mid f)^+ gh$
3	$(a \mid b \mid c \mid d)^* \mid e$	$(a \mid b \mid c \mid d)^* \mid e$	$(e(a \mid c \mid d \mid b)^+ e)$
4	$(abcde)^* f$	$(abcde)^* f$	$(f(a \mid e \mid d \mid c \mid b)^+ f)$
5	$(ab)^* \mid cdef \mid (ghi)^*$	$(ab)^* \mid cdef \mid (ghi)^*$	$cdef(a \mid b \mid g \mid i \mid h)^+ cdef$
6	$abcdef(g \mid h \mid i \mid j)(k \mid l \mid m \mid n \mid o)$	$abcdef(g \mid h \mid i \mid j)(k \mid l \mid m \mid n \mid o)$	$abcdef(j(o \mid l \mid m \mid n \mid k) \mid g(o \mid l \mid n \mid m \mid k) \mid$ $h(m \mid l \mid n \mid k \mid o) \mid i(o \mid l \mid n \mid m \mid k))$
7	$(a \mid b \mid c)d^* e^* (fgh)^*$	$(a \mid b \mid c)d^* e^* (fgh)^*$	$((c \mid b \mid a)d^+ e^+ \mid ad^+ \mid bd^+ \mid c(e^+ \mid d^+)? \mid$ $ad^* \mid be^*))(f \mid h \mid g)^+ ((a \mid b \mid c)d^+ e^+ \mid$ $c(e^+ \mid d^+)? \mid a(e^+ \mid d^+)? \mid b(e^+ \mid d^+)?)$
8	$(a \mid b)(cdefg)^*$ $hijklmnopq(r \mid s)^*$	$(a \mid b)(cdefg)^*$ $hijklmnopq(r \mid s)^*$	$((((a \mid b)hijabcdefg) \mid b \mid a)$ $(c \mid g \mid f \mid e \mid d \mid s \mid r)^+ ((b \mid a)?hijkamnopq))$
9	$(abcd)^* \mid (e \mid f \mid g)^* \mid h \mid (ijklm)^*$	$(abcd)^* \mid (ijklm)^* \mid h \mid (e \mid f \mid g)^*$	$h(a \mid d \mid c \mid b \mid e \mid g \mid f \mid i \mid m \mid l \mid k \mid j)^+ h$
10	$a^* \mid (b \mid c \mid d \mid e \mid f)^* \mid gh \mid (i \mid j \mid k)^* \mid$ $(lmn)^*$	$a^* \mid (b \mid c \mid d \mid e \mid f)^* \mid gh \mid (i \mid j \mid k)^* \mid$ $(lmn)^*$	$(a^+ \mid gh)(e \mid f \mid d \mid i \mid j \mid l \mid n \mid m \mid k \mid c \mid b)^+$ $(a^+ \mid gh)$

Table 5: DTDs generated by XTRACT and DDbE for Synthetic Data Set

NO	Simplified DTD	DTD Obtained by XTRACT	DTD obtained by DDbE
1	$a \mid b \mid c \mid d \mid e$	$a \mid b \mid c \mid d \mid e$	$a \mid b \mid c \mid d \mid e$
2	$(a \mid b \mid c \mid d \mid e)^*$	$(a \mid b \mid c \mid d \mid e)^*$	$(a \mid b \mid c \mid d \mid e)^*$
3	$(ab^* c^*)$	$ab^* c^*$	$(ab^+ c^*) \mid (ac^*)$
4	$a^* b?c?d?$	$a^* b?c?d?$	$(a^+ b(c \mid (c?d))?) \mid ((b \mid a^+)?cd) \mid$ $((a^+ \mid b)?d) \mid ((a^+ \mid b)?c) \mid a^+ \mid b)$
5	$(a(bc)^+ d)*$	$(a(bc)^* d)^*$	$(a \mid b \mid c \mid d)^+$
6	$(ab?c^* d?)*$	–	$(a \mid b \mid c \mid d)+$

Table 6: DTDs generated by XTRACT and DDbE for Real-life Data Set

neither XTRACT nor DDbE is able to correctly infer DTD 6. (The approximate DTD derived by XTRACT for DTD 6 is rather complex and, therefore, we chose to omit it from Table 6.) The reason for XTRACT's failure is that our generalization subsystem does not detect patterns containing the optional symbol ?. Finding such patterns requires a more sophisticated analysis of symbol occurrences within and across sequences, and we plan to pursue this further as part of future work.

8 Conclusions

In this paper, we presented the architecture of the XTRACT system for inferring a DTD for a database of XML documents. The DTD plays the role of a schema and thus contains valuable information about the structure of the XML documents that it describes. However, since DTDs are *not mandatory*, in a number of cases, documents in an XML database may not have an accompanying DTD. Thus, the DTD inference problem is important, especially given the critical role that the DTD plays in the storage as well as the formulation, optimization and processing of queries on the underlying data.

The problem of deriving the DTD for a set of documents is complicated by the fact that the DTD syntax incorporates the full expressive power of regular expressions. Specifically, as we showed, naive approaches that do not "generalize" beyond the input element sequences fail to deduce concise and semantically meaningful DTDs. Instead, XTRACT applies sophisticated algorithms in three steps to compute a DTD that is more along the lines that a human would infer. In the first *generalization* step, patterns

within the input sequences are detected and more "general" regular expressions are substituted for them. These "generalized" candidate DTDs are then processed by the *factorization* step that factors common expressions within the DTDs to make them more succinct. The first two steps thus produce a range of candidate DTDs that vary in their conciseness and precision. In the third and final step, XTRACT employs the MDL principle to select from amongst the candidates the DTD that strikes the right balance between conciseness and preciseness – that is, a DTD that is concise, but at the same time, is not too general. The MDL principle maps naturally to the *facility location problem* (FLP) which we solved using an efficient approximation algorithm recently proposed in the literature.

We compared the quality of the DTDs inferred by XTRACT with those returned by the IBM alphaworks DDbE (Data Descriptors by Example) DTD extraction tool on synthetic as well as real-life DTDs. In our experiments, XTRACT outperformed DDbE by a wide margin, and for most DTDs it was able to accurately infer the DTD while DDbE completely failed to do so. A number of the DTDs which were correctly identified by XTRACT were fairly complex and contained factors, metacharacters and nested regular expression terms. Thus, our results clearly demonstrate the effectiveness of XTRACT's approach that employs generalization and factorization to derive a range of general and concise candidate DTDs, and then uses the MDL principle as the basis to select amongst them. While we are encouraged by XTRACT's performance, we are continuing to further enhance our algorithms to infer even more complex DTDs (than those considered in this paper).

References

[Abi97] S. Abiteboul. Querying semi-structured data. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 1–18, 1997.

[Ang78] Dana Angluin. On the complexity of minimum inference of regular sets. *Information and Control*, 39(3):337–350, 1978.

[BM82] R. K. Brayton and C. McMullen. The decomposition and factorization of boolean expressions. In *International Symposium on Circuits and Systems*, pages 49–54, 1982.

[BPSM] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML). http://www.w3.org/TR/REC-xml.

[Bra93] Alvis Brazma. Efficient identification of regular expressions from representative examples. *COLT*, pages 236–242, 1993.

[CG99] M. Charikar and S. Guha. Improved combinatorial algorithms for the facility location and k-median problems. In *40th Annual Symposium on Foundations of Computer Science*, 1999.

[DFS99] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with stored. In *Proc. of the ACM SIGMOD Conference on Management of Data*, June 1999.

[FS97] M. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proceedings of the International Conference on Database Theory (ICDT)*, 1997.

[GMW99] R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the lore data model and query language. In *Proceedings of the 2nd International Workshop on the Web and Databases (WebDB '99)*, pages 25–30, June 1999.

[Gol67] E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[Gol78] E. Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.

[GW97] R. Goldman and J. Widom. "DataGuides: Enabling query formulation and optimization in semistructured databases". In *Proceedings of the 23rd International Conference on Very Large Data Bases*, Athens, Greece, August 1997.

[Hoc82] D. S. Hochbaum. Heuristics for the fixed cost median problem. *Mathematical Programming*, 22:148–162, 1982.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automaton Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.

[KMU95] P. Kilpel"ainen, H. Mannila, and E. Ukkonen. MDL learning of unions of simple pattern languages from positive examples. In *Second European Conference on Computational Learning Theory, EuroCOLT*, pages 252–260, Barcelona, 1995.

[Law64] E. Lawler. "An approach to multilevel boolean minimization". *Journal of the ACM*, July 1964.

[MRA95] Manish Mehta, Jorma Rissanen, and Rakesh Agrawal. MDL-based decision tree pruning. In *Int'l Conference on Knowledge Discovery in Databases and Data Mining (KDD-95)*, Montreal, Canada, August 1995.

[NAM98] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 295–306, 1998.

[Pit89] L. Pitt. "Inductive inference, DFAs, and computational complexity". *Analogical and Inductive Inference*, pages 18–44, 1989.

[QR89a] J. R. Quinlan and R. L. Rivest. Inferring decision trees using minimum description length principle. *Information and Computation*, 1989.

[QR89b] J. Ross Quinlan and Ronald L. Rivest. "Inferring Decision Trees Using the Minimum Description Length Principle". *Information and Computation*, 80:227–248, 1989.

[Ris78] J. Rissanen. Modeling by shortest data description. *Automatica*, 14:465–471, 1978.

[Ris89] J. Rissanen. *Stochastic Complexity in Statistical Inquiry*. World Scientific Publ. Co., 1989.

[SHT+99] J. Shanmugasundaram, G. He, K. Tufte, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. of the Int'l Conf. on Very Large Data Bases*, Edinburgh, Scotland, 1999.

[Wan89] A. R. R. Wang. *Algorithms for Multi-level Logic Optimization*. PhD thesis, The University of California, Berkeley, 1989.

[Wid91] J. Widom. Data management for XML: research directions. *IEEE Data Engineering Bulletin*, 1991.

– Appendix –

```
procedure FACTOR(S)        /* S is the set of sequences to be factored */
begin
    DivisorSet := FINDALLDIVISORS(S)
    if (DivisorSet = φ)
        return or of sequences in S
    DivisorList := φ
    for each divisor V in DivisorSet
        Q, R := DIVIDE(S, V)
        add (V, Q, R) to DivisorList
    find the most compact triplet (V_i, Q_i, R_i) in DivisorList
    return (FACTOR(V_i))(FACTOR(Q_i)) | FACTOR(R_i)
end

procedure FINDALLDIVISORS(S)
begin
    DivisorSet := φ
    for each distinct sequence s such that s is a suffix for at least two elements in S
        DivisorSet := DivisorSet ∪ {{p : ps ∈ S}}
    return DivisorSet
end

procedure DIVIDE(S, V)
begin
    for each sequence p in V
        q_p := {s : ps ∈ S}
    Q := ∩_{p∈V} q_p
    R := S − V ∘ Q
    /* V ∘ Q is the set of sequences resulting from concatenating
        every sequence in Q to the end of every sequence in V */
    return Q, R
end
```

Figure 7: Factoring Algorithm

EXHIBIT B

April 28, 2000
Draft of Application

TITLE: DOCUMENT DESCRIPTOR EXTRACTION METHOD

FIELD OF THE INVENTION

The present invention relates to electronic documents. Specifically, the present invention relates to determining document descriptors from data within electronic documents.

BACKGROUND OF THE INVENTION

The number of documents available in electronic format has exploded. With the number of available electronic documents increasing rapidly, it is important to be able to quickly and accurately search the available electronic documents. In addition, it is desirable to be able to store data into electronic documents and generate new electronic documents which are similar in structure to existing electronic documents. Hence, tools which assist in the querying of electronic documents, the creation of electronic documents, and the storage of data into electronic documents are desirable.

Electronic documents for display over the Internet and/or an Intranet are commonly stored in a Standard Generalized Markup Language (SGML) format. SGML is a standard for how to specify a document markup language or tag set. SGML is not in itself a document language, but a description of how to specify one. The SGML format provides for the inclusion of a document type descriptor (DTD). A document's DTD specifies how the data within a document should be organized. One SGML format for storing data within electronic documents which is becoming increasingly popular is eXtensible Markup Language (XML). XML is rapidly emerging as the new standard for representing and exchanging data on the World Wide Web (web). An XML document may be accompanied by a document type descriptor (DTD). For example, in an XML document, the DTD may specify the tags which can be used, the order in which the tags appear, how the tags are nested, and tag attributes.

Thus, the DTD plays an important role in the storage of data to the XML document, the generation of similar documents, and increasing the efficiency of queries of the XML document. Efficiency is achieved by using the knowledge of the structure of the data to remove elements that cannot potentially satisfy the query.

Although DTDs are helpful in the storage, generation, and retrieval of data related to an XML document, DTDs are not mandatory. Since DTDs are not mandatory, many XML documents exist which do not contain DTDs. In addition, since only a small portion of the electronic documents in existence today are in an XML format, initially the majority of XML documents will likely be automatically generated from pre-existing non-XML documents. In many instances, the automatically generated XML formatted documents will not contain DTDs. Therefore, a tool for automatically generating DTDs is desirable for improving data storage and retrieval.

Others have attempted to automatically generate DTDs with varying degrees of success. One system is IBM's Data Descriptors by Example (DDbE) system. The goal of DDbE is to give users a good start at creating DTDs for their own applications. However, this system and other available systems do not produce highly accurate DTDs for all XML documents, especially complex XML documents. Since accurate DTDs enable efficient storage and retrieval of data, improved methods for extracting accurate DTDs from XML documents are desirable.

SUMMARY OF THE INVENTION

The present invention relates to developing a description of the layout of an electronic document from data within the document. The present invention is especially useful for

determining document type descriptors (DTDs) of electronic documents in a Standard Generalized Markup Language (SGML) format.

The present invention comprises generalizing input sequences generated from an electronic document. The input sequence are generalized to create generalized sequences which are representative of the input sequences. Each generalized sequence encompasses one or more input sequences in a more general form. Next, the present invention comprises selecting a description of the layout of the electronic document from the input sequences and generalized sequences. Selecting a description comprises selecting one or more of the input sequences and generalized sequences such that every input sequence is encompassed by the selected sequences. Preferably, the selection is performed using minimum descriptor length (MDL) principles.

Additionally, the present invention may comprise factoring the input sequences and generalized sequences after generalizing to create factored sequences which can be included in the selection of the description. Each factored sequence encompasses one or more input sequences and generalized sequences. The factored sequence are combined with the input sequences and generalized sequences, thereby creating a potentially better selection of sequences from which a description may be selected.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is a flow chart of a preferred document type descriptor (DTD) extraction system in accordance with the present invention; and

Figure 2 is an illustrative depiction of the output of each step and the selection process of the preferred document type descriptor extraction system depicted in Figure 1 in accordance with the present invention.

DETAILED DESCRIPTION OF THE INVENTION

The present invention relates to inferring (i.e., determining) document descriptors from data within electronic documents. For illustrative purposes, the present invention is described in terms of inferring document type descriptors (DTDs) from data within eXtensible Markup Language (XML) formatted documents. However, it will be readily apparent to those skilled in the art that the present invention could be applied to other types of markup languages which provide document descriptions that are currently available or developed in the future, such as markup languages which conform to the Standard General Markup Language (SGML) format. The inferred DTD contains valuable information about the structure of the XML documents that it describes. The structural information may be used to efficiently query the XML document, store data to the XML document, or generate similar XML documents.

A sample XML document and its associated DTD are as follows:

Sample XML Document

```
<article>
   <title> A Relational Model for Large Shared Data Banks </title>
   <author>
       <name> E. F. Codd </name>
       <affiliation> IBM Research </affiliation>
   </author>
</article>
<article>
   <title> XTRACT: A system for Extracting DTDs </title>
   <author>
       <name> M. Garofalakis </name>
       <affiliation> Bell Labs </affiliation>
   </author>
   <author>
       <name> A. Gionis </name>
       <affiliation> Stanford University </affiliation>
   </author>
   <author>
       <name> R. Rastogi </name>
```

```
        <affiliation> Bell Labs </affiliation>
    </author>
    <author>
        <name> S. Seshadri </name>
        <affiliation> Bell Labs </affiliation>
    </author>
    <author>
        <name> K. Shim </name>
        <affiliation> Bell Labs </affiliation>
    </author>
</article>
```

Sample Document Type Descriptor (DTD)

```
<!ELEMENT article (title, author*)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (name, affiliation)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT affiliation (#PCDATA)>
```

A DTD describes the structure of an XML document. A DTD constrains the structure of an element by specifying a regular expression with which its sub-element sequences must conform. The DTD declaration sequence uses commas for sequencing, | for exclusive OR, parenthesis for grouping and meta-characters ?, *, + to denote zero or one, zero or more, and one or more, respectively.

In the sample XML document above and its associated DTD, the start of an element such as article is indicated by <article> and the end of the element is indicated by </article>. Each element may comprise sub-elements and/or data. For example, for the element article, title and author are sub-elements. Likewise, sub-elements may further contain additional sub-elements. For example, author contains sub-element name and sub-element affiliation.

In a preferred embodiment, the present invention applies algorithms in three steps to compute a DTD from a set of input sequences. They are (1) generalizing, (2) factoring, and (3) selecting.

The input sequences are groupings of sub-elements contained within each occurrence of an element. For an element, such as article, it is straight forward to compute the sequence of sub-elements nested within each <article> </article> pair in the XML document. The set of input sequences comprises one sequence for each occurrence of element <article>. For example, in the above XML document sample the input sequences for <article> would be input sequences <title><author> and <title><author><author><author><author><author>. For ease of description, the first letter of the sub-element may be used as a shorthand for describing sequences (e.g., <title> <author> is represented by ta and <title><author><author><author><author><author> is represented by taaaaa.)

In the preferred embodiment, the input sequences are generalized to create generalized sequences. The generalized sequences and input sequences are then factored to create factored sequences. Each factored sequence may encompass one or more input sequences and generalized sequences, thereby creating additional sequences which may be selected as a part of a DTD. The factoring step is optional. However, using the factoring step results in potentially better DTDs. Factoring leads to better DTDs by creating additional sequences from which an appropriate DTD may be selected. A DTD which encompasses all of the input sequences is then selected from the input sequences, generalized sequences, and factored sequences.

In the generalization step, patterns within the input sequences are detected and more "general" regular expressions are substituted for them to create "generalized" sequences. In a preferred embodiment, the "generalized" sequences and the input sequences are then processed by the factorization step which factors common expressions to make them more succinct. The factorization step yields "factored" sequences. The first two steps along with the input sequences produce a series of potential DTDs that vary in their conciseness and

precision. A selection step then selects a DTD from the candidates that strikes the right balance between conciseness and preciseness - that is, a DTD that is concise, but at the same time, is not too general. In a preferred embodiment, the selection step employs minimum descriptor length (MDL) principles for selecting a DTD.

Figure 1 depicts a flow chart 100 illustrating the steps for inferring a DTD in accordance with a preferred embodiment of the present invention. The input sequences I are comprised of sub-elements a, b, c, d, and e. The input sequences are first processed by a generalization module 110 which produces generalized sequences. The generalized sequences are combined with the input sequences to create a set of potential DTDs identified by S_G. Optionally, the potential DTDs are factored using a factoring module 120. The factoring module produces additional potential DTDs which are combined with the potential DTDs output by the generalization module 110 to create a set of potential DTDs identified by S_F. Finally, the selecting module 130 infers (i.e. selects) a DTD from all of the potential DTDs S_F. Preferably, the selecting module 130 incorporates MDL principles.

Figure 2 graphically depicts the selection of a DTD from all of the potential DTDs. The selected DTD must encompass all of the original input sequences S_F. It can be seen that (ab)* encompasses input sequences ab and abab. Also, (a|b) (c|d) encompasses input sequences ac, ad, bc, and bd. Finally, b*(d|e) encompasses bd, bbd, and bbbbe. The selected potential DTDs, when combined using ORs, encompass all of the original input sequences. The result is a concise and precise DTD.

I. <u>GENERALIZING</u>

The quality of the data type descriptor (DTD) selected during the selection process is very dependent on the set of candidate DTDs available. If the selection were based on the

input sequences only, then the final DTD output by the selection step would simply be the OR of all the input sequences. For example, in the above XML document sample, the DTD for <article> would comprise ta and taaaa (i.e., <title><author> and <title><author><author><author><author><author>.) However, this is not a desirable DTD since it is neither concise nor intuitive. A more concise and intuitive DTD would be the single sequence ta* which encompasses both ta and taaaaa. Thus, in order to infer meaningful DTDs, the candidate DTDs should be general. Ideally, each candidate DTD encompasses more than one input sequence. The goal of the generalization module 110 is to achieve this objective.

The generalization module 110 of the present invention infers a number of regular expressions which have been found to frequently appear in real-life DTDs. Below, are examples of regular expressions from real-life DTDs that appear in the Newspaper Association of America (NAA) Classified Advertizing Standards XML DTD (found at http://www.naa.org/technology/clsstdtf/Adex010.dtd).

a* bc* : DTDs of this form are generally used to specify tuples with set-valued attributes.
<!ELEMENT account-info (account-number, sub-account-number*)> <!-- Specification for account identification information -->

(abc)* : This type of DTD is used to represent a set (or a list) of ordered tuples.
<!ELEMENT days-and-hours (date, time)+> <!-- provide times/dates when job fairs will be held -->

(a|b|c)* : The DTD of the form (a|b|c)* is frequently used to represent a multiset containing the elements a, b and c. This DTD is very useful since the elements in the multiset are allowed to appear multiple times and in any order in the document. For example, the following DTD specifies that the support information for an ad can consist of an arbitrary number of audio or video clips, photos, and further these can appear in any order.
<!ELEMENT support-info (audio-clip | file-id | graphic | logo | new-list | photo | video-clip | zz-generic-tag)*> <!-- support information for ad content -->

((ab)* c)* : This type of DTD permits nesting relationships among sets (OR lists).

<!ELEMENT transfer-info (transfer-number, (from-to, company-id)+, contact-info)*>
<!-- provides parent information through the multilevel aggregation process. may be repeated -->

Table 1 depicts pseudo code for a preferred generalization algorithm (Procedure GENERALIZE). Procedure GENERALIZE infers several DTDs for each input sequence independently and adds them to the set S_G. The generalize algorithm may over-generalize in some cases (since DTDs are inferred based on a single sequence), however, the selection step in selecting module 130 will ensure that such overly-general DTDs are not chosen as part of the final inferred DTD, if there are better alternatives. The generalization step will provide several alternate candidates in addition to the input sequences for the selection step.

The algorithm can infer regular expressions that are more complex than the above, however, complex expressions, such as (ab?c* d?)*, that are less likely to occur in practice, may be excluded.

procedure GENERALIZE(I)
begin
1. **for each** sequence s in I
2. add s to S_g
3. **for** r := 2, 3, 4
4. s' := DISCOVERSEQPATTERN(s, r)
5. **for** $d := 0.1 \cdot |s'|, 0.5 \cdot |s'|, |s'|$
6. s'' := DISCOVERORPATTERN(s', d)
7. add s'' to S_g
end

procedure DISCOVERSEQPATTERN(s, r)
begin
1. **repeat**
2. let χ be a subsequence of s with the maximum number ($\geq r$) of contiguous repetitions in s
3. replace all ($\geq r$) contiguous occurrences of χ in s with a new auxiliary symbol $A_i = (\chi)^*$
4. **until** (s no longer contains $\geq r$ contiguous occurrences of any subsequence χ)
5. **return** s
end

procedure DISCOVERSEQPATTERN(s, d)

```
begin
1. s1, s2,..., sn := PARTITION(s, d)
2.   for each subsequence sj in s1, s2..., sn
3.      let the set of distinct symbols in sj be a1, a2,..., am
4.      if (m>1)
5.            replace subsequence sj in sequence s by a new auxiliary symbol Ai =
              (a1/···/am)*
6. return s
end

procedure PARTITION(s, d)
begin
1. i := start := end := 1
2. si := s[start, end]
3. while (end < |s|)
4.      while (end < |s| and a symbol in si occurs to the right of si within a distance d)
5.            end := end + 1; si := s[start, end]
6.      if (end < |s|)
7.            i := i + 1; start := end + 1; end := end + 1; si := s[start, end]
8. return s1, s2···, si
end
```

Table 1: Generalization Algorithm

The essence of procedure GENERALIZE are the procedures DISCOVERSEQPATTERN and DISCOVERORPATTERN which are repeatedly called with predefined parameter values.

Discovering Sequencing Patterns (Procedure DISCOVERSEQPATTERN)

Procedure DISCOVERSEQPATTERN, shown in Table 1, takes an input sequence s and returns a candidate DTD that is derived from s by replacing sequencing patterns of the form xx... x, for a subsequence x in s, with the regular expression (x)*. In addition to s, the procedure also accepts as input, a threshold parameter r > 1 which is the minimum number of contiguous repetitions of subsequence x in s required for the repetitions to be replaced with (x)*. In case there are multiple subsequences x with the maximum number of repetitions in

step 2 of procedure DISCOVERSEQPATTERNS, the longest among them is chosen, and subsequent ties are resolved arbitrarily.

Note that instead of introducing the regular expression term (x)* into the sequence s, an auxiliary symbol that serves as a representative for the term is introduced. The use of auxiliary symbols enable the description of the algorithms to remain simple and clean since the input to them is always a sequence of symbols. In a preferred embodiment, there is a one-to-one correspondence between auxiliary symbols and regular expression terms in the present invention; thus, if the auxilliary symbol A denotes (bc)* in one candidate DTD, then it represents (bc)* in every other candidate DTD. Also, procedure DISCOVERSEQPATTERN may perform several iterations and thus new sequencing patterns may contain auxiliary symbols corresponding to patterns replaced in previous iterations. For example, invoking procedure DISCOVERSEQPATTERN with the input sequence s = abababcababc and r = 2 yields the sequence A_1cA_1c after the first iteration, where A_1 is an auxiliary symbol for the term (ab)*. After the second iteration, the procedure returns the candidate DTD A_2, where A_2 is the auxiliary symbol corresponding to ((ab)* c)*. Thus, the resulting candidate DTD returned by procedure DISCOVERSEQPATTERN can contain *s nested within other *s. Finally, DISCOVERSEQPATTERN is invoked with three different values for the parameter r to control the aggressiveness of the generalization. For example, for the sequence aabbb, DISCOVERSEQPATTERN with r = 2 would infer a* b*, while with r = 3, it would infer aab*. In the selection step, if many other sequences are encompassed by aab*, then a DTD of aab* may be preferred to a DTD of a* b* since it more accurately describes the input sequences.

Discovering OR Patterns (Procedure DISCOVERORPATTERN)

Procedure DISCOVERORPATTERN, shown in Table 1, infers patterns of the form $(a_1|a_2| \dots |a_m)^*$ based on the locality of these symbols within a sequence s. The locality is identified by first partitioning (performed by procedure PARTITION, shown in Table 1) the input sequence s into the smallest possible subsequences $s_1, s_2, \dots, s_n$, such that for any occurrence of a symbol a in a subsequence s_i, there does not exist another occurrence of a in some other subsequence s_j within a distance d (which is a parameter to DISCOVERORPATTERN). Each subsequence s_i in s is then replaced by the pattern $(a_1|a_2| \dots |a_m)^*$ where $a_1, \dots, a_m$ are the distinct symbols in the subsequence s_i. If s_i contains frequent repetitions of the symbols $a_1|a_2|\dots|a_m$ in close proximity, then it is very likely that s_i originated from a regular expression of the form $(a_1|a_2|\dots|a_m)^*$. For illustrative purposes, for the input sequence abcbac, procedure DISCOVERORPATTERN returns:

- aA_1ac for d = 2, where $A_1 = (b \mid c)^*$;
- aA_2 for d = 3, where $A_2 = (a \mid b \mid c)^*$; and
- A_2 for d = 4, where $A_2 = (a \mid b \mid c)^*$.

A preferred component for discovering OR patterns is procedure PARTITION, shown in Table 1. For a sequence s, s[i,j] denotes the subsequence of s starting at the i^{th} symbol and ending at the j^{th} symbol of s. Procedure PARTITION constructs the subsequences in the order s_1, s_2, and so on. Assuming that s_1 through s_j have been generated, it constructs s_{j+1} by starting s_{j+1} immediately after s_j ends and expanding the subsequence s_{j+1} to the right as long as required to ensure that there is no symbol in s_{j+1} that occurs within a distance d to the right of s_{j+1}. By construction, there cannot exist such a symbol to the left of s_{j+1}.

Note that procedure GENERALIZE invokes DISCOVERORPATTERN on the DTDs that result from calls to DISCOVERSEQPATTERN and therefore it is possible to infer more complex DTDs of the form $(a|(bc)^*)^*$ in addition to DTDs like $(a|b|c)^*$. For instance, for the

input sequence s = abcbca, procedure DISCOVERSEQPATTERN invoked with r = 2 would return s' = aA_1a, where $A_1 = (bc)^*$, which, when input to DISCOVERORPATTERN returns s'' = A_2 for d = |s'|, where $A_2 = (a|A_1)^*$. Further, DISCOVERORPATTERN is invoked with various values of d (expressed as a fraction of the length of the input sequence) to control the degree of generalization. Small values of d lead to conservative generalizations while larger values result in more liberal generalizations. The size of d is based on desired design characteristics.

II. FACTORING

In a preferred embodiment, the factoring module 120 uses a factoring step to derive factored forms for expressions that are an OR of a subset of the candidate DTDs, S_G, out of the generalization module 110. For example, for candidate DTDs ac, ad, bc and bd in S_G, the factoring step would generate the factored form (a | b)(c | d). Note that since the final DTD is an OR of candidate DTDs, S_F, out of the factoring module 120, the factored forms are also candidates. Further, a factored candidate DTD, because of its smaller size, has a lower minimum description length (MDL) cost, and is thus more likely to be chosen in the selection step, if MDL principles are used. Thus, since factored forms (due to their compactness) are more desirable, factoring can result in better quality DTDs.

Factored DTDs are common in real life. For example, in the sample DTD, an article may be categorized based on whether it appeared in a workshop, conference or journal; it may also be classified according to its area as belonging to either computer science, physics, chemistry etc. Thus, the DTD (in factored form) for the element article would be as follows:

<!ELEMENT article(title, author*, (workshop | conference | journal), (computer science | physics | chemistry | ...))

The set of candidate DTDs, S_F, output by the factorization module, 120, in addition to the factored forms generated from candidates in S_G, also contains all the DTDs in S_G. Ideally, factored forms for every subset of S_G, should be added to S_F to be considered by the selection step. However, this may be impractical, since S_G could be very large. Therefore, a heuristic may be used to select subsets of candidates in S_G that when factored yield good factored DTDs. In a preferred embodiment, the factoring algorithm is an adaptation of factoring algorithms for boolean expressions which are well known in the art.

Selecting Subsets of S_G to Factor

Intuitively, a subset S of S_G out of generalization module 110 is a good candidate for factoring if the factored form of S is much smaller than S itself. In addition, even though S_G may contain multiple generalizations that are derived from the same input sequence, it is highly unlikely that the final DTD will contain two generalizations of the same input sequence. Thus, factoring candidate DTDs in S_G that encompass similar sets of input sequences does not lead to factors that can improve the quality of the final DTD.

For a subset S of S_G to yield good factored forms it must satisfy the following two properties:

(1.) Every DTD in S has a common prefix or suffix with a number of other DTDs in S. Further, as more DTDs in S share common prefixes or suffixes, or as the length of the common prefixes/suffixes increases, the quality of the generated factored form can be expected to improve.

(2.) The overlap between every pair of DTDs D; D' in S is minimal, that is, the intersection of the input sequences encompassed by D and D' is small. This is important because, as mentioned above, a factored DTD adds little value (from an MDL cost perspective) over the candidate DTDs from which it was derived if it cannot be used to encode a significantly larger number of input sequences compared to the sequences encompassed by each individual DTD.

In order to state properties (1) and (2) for a set S of DTDs more formally. The following notation is used. For a DTD D, let cover(D) denote the input sequences in I that are encompassed by D (note that auxiliary symbols are expanded completely when cover for a DTD is computed). Then, overlap(D, D') is defined as the fraction of the input sequences encompassed by D and D' that are common to D and D', that is,

(1)

Thus, for a sufficiently small value of a (user-specified) parameter δ, by ensuring that overlap(D,D') < δ for every pair of DTDs D and D' in S, it can be ensured that S satisfies property (2) mentioned above.

In order to characterize property (1) more rigorously, the function score(D,S) is introduced in equation 2. Function score (D, S) attempts to capture the degree of similarity between prefixes/suffixes of DTD D and those of DTDs in the set S of DTDs. Intuitively, a DTD with a high score with respect to set S is a good candidate to be factored with other DTDs in set S. For a DTD D, let pref (D) and suf(D) denote the set of prefixes and suffixes of D, respectively. Let psup(p,S) denote the support of prefix p in set S of DTDs, that is, the number of DTDs in S for which p is a prefix. Similarly, let ssup(s,S) denote number of DTDs in S for which s is a suffix. Then score(D,S) is defined as follows:

$$\mathrm{score}(D,S) = \max(\{|p| \,.\, \mathrm{psup}(p,S) : p \in \mathrm{pref}(D)\} \cup \{|s| * \mathrm{ssup}(s,S) : s \in \mathrm{suf}(D)\}) \quad (2)$$

Thus, the prefix/suffix p=s of D, for which the product of p=s's length and its support in S is maximum, determines the score of D with respect to S. If DTD D has a long prefix or suffix that occurs frequently in set S, then this prefix can be factored out, thus resulting in good factored forms. The function score is thus a good measure of how well D would factor with other DTDs in S.

Procedure FACTORSUBSETS, shown in Table 2, first selects subsets S of sequences from within sequences S_G that satisfy properties (1) and (2). Each of these subsets S is then factored by invoking procedure FACTOR (in Step 15), depicted in Table 3. Assuming that the factoring algorithm returns $F_1 \mid F_2 \mid \ldots\ F_m$, each of the F_i is added to S_F.

procedure FACTORSUBSETS(S_g)
begin
1. **for each** DTD D is S_g
2.   Compute *score* (D,S_g)
3. $S_F := S' := S_g$; SeedSet := $\emptyset$
4. **for** $i := 1$ to k
5.   let D be the DTD in S' with the maximum value for *score* (D,S_g)
6.   SeedSet := SeedSet $\cup$ D
7. $S' := S' - \{D' : overlap\ (D, D') \geq \delta\}$
8. **for each** DTD D in SeedSet
9. $S := \{D\}$
10. $S' := S_g - \{D' : overlap\ (D, D') \geq \delta\}$
11.   **while** (S' is not empty)
12.   let D' be the DTD in S' with the maximum value for *score* (D',S)
13. $S := S \cup D'$
14. $S' := S' - \{D' : overlap\ (D', D'') \geq \delta\}$
15.   F := FACTOR(S)
16. $S_F := S_F \cup \{F_1;\ldots,F_m\}$ /* $F = F_1 | \cdots | F_m$ */
end

Table 2: Choosing Subsets Of S_g For Factoring

Procedure FACTORSUBSETS computes a set S of candidate DTDs to factor. First, k seed DTDs for the sets S to be factored are chosen in the for loop spanning steps 4-7. These seed DTDs have a high score value with respect to S_G and overlap minimally with each other. Thus, it is ensured that each seed DTD not only factors well with other DTDs in S_G, but is also significantly different from other seeds. In steps 9-14, each seed DTD is used to construct a new set S of DTDs to be factored (thus, only k sets of DTDs are generated). After initializing S to a seed DTD D, in each subsequent iteration, the next DTD D' that is added to

S is chosen greedily (i.e., the one whose score with respect to DTDs in S is maximum and whose overlap with DTDs already in S is less than δ).

Algorithm For Factoring a Set of DTDs

Algorithms for computing the optimum factored form, that is, the one with the minimum number of literals are known in the art. However, the complexity of these known techniques may be impractical. In a preferred embodiment, heuristic factoring algorithms for boolean functions which are known in the art are adapted for use in the present invention. Factored forms of boolean functions are commonly used in VLSI design.

There is a close correspondence between the semantics of DTDs and those of boolean expressions. The sequencing operator (,) in DTDs is similar to a logical AND in boolean algebra, while the OR operator (|) is like a logical OR. However, there exist certain fundamental differences between DTDs and boolean expressions. First, while the logical AND operator in boolean logic is commutative, the sequencing operator in DTDs is not (the ordering of symbols in a sequence matters!). Second, in boolean logic, the expression a | ab is equivalent to a; however, the equivalent DTD for a | ab is ab?. The boolean algorithms can be modified to create a factoring algorithm to handle the semantics of the DTDs. The pseudo-code for procedure FACTOR, is shown in Table 3. Procedure FACTOR is a preferred embodiment of the factoring algorithm used in factoring module 120.

procedure FACTOR(*S*)   /* *S* is the set of sequences to be factored */
begin
1. DivisorSet := FINDALLDIVISORS(*S*)

2. **if** (DivisorSet = ⊘)
3. **return** *or* of sequences in S
4. DivisorList := ⊘
5. **for each** divisor V in DivisorSet
6. Q, R := DIVIDE(S, V)
7. add (V, Q, R) to DivisorList
8. find the most compact triplet (V_i Q_i R_i) in DivisorList
9. **return** (FACTOR(V_i))(FACTOR(Q_i)) | FACTOR(R_i)

end

procedure FINDALLDIVISORS(S)
begin
1. DivisorSet := ⊘
2. **for each** distinct sequence s such that s is a suffix for at least two elements in S
3. DivisorSet := DivisorSet ∪ $\{\{p : ps \in S\}\}$
4. **return** DivisorSet

end

procedure DIVIDE(S, V)
begin
1. **for each** sequence p and V
2. $q_p := \{s : ps \in S\}$
3. $Q := \cap_{p \in v} q_p$
4. $R := S - V \circ Q$
/* $V \circ Q$ is the set of sequences resulting from concatenating every sequence in Q to the end of every sequence in V */
5. **return** Q, R

end

Table 3: Factoring Algorithm

As an example of the factoring algorithm, consider the set S = {b, c, ab, ac, df, dg, ef, eg} of input sequences corresponding to the expression b|c|ab|ac|df|dg|ef|eg whose factored form is a?(b|c)|(d|e)(f|g). Before the steps that procedure FACTOR performs to derive the factored form are discussed, the DIVIDE operation that constitutes the core of the factoring algorithm is introduced. For sets of sequences S, V, DIVIDE(S,V) returns a quotient Q and remainder V such that S = V ∘ Q ∪ R (here, V ∘ Q is the set of sequences resulting from concatenating every sequence in Q to the end of every sequence in V). Thus, for the above

set S and V = {d,e}, DIVIDE(S,V) returns the quotient Q = {f,g} and remainder R = {b,c,ab,ac}. The steps executed by FACTOR to generate the factored form are as follows:

(1.) Compute set of potential divisors for S. These are simply sets of prefixes that have a common suffix in S. Thus, potential divisors for S include {d, e} (both f and g are common suffixes) and {1,a} (both b and c are common suffixes). The symbol "1" is special and denotes the identity symbol with respect to the sequencing operator, that is, 1s = s1 = s for every sequence s.

(2.) Choose divisor V from set of potential divisors. This is carried out by first dividing S by each potential divisor V to obtain a quotient Q and remainder R, and then selecting the V for which the triplet (V,Q,R) has the smallest size. In our case, V = {d,e} results in a smaller quotient and remainder (Q = {f, g}, R = {b, c, ab, ac}) than {1,a} (Q = {b,c}, R = {df,dg,ef,eg}) and is thus chosen.

(3.) Recursively factor V, Q, and R. The final factored form is FACTOR(V)FACTOR(Q)|FACTOR(R), where V = {d|e}, Q = {f,g} and R = {b,c,ab,ac}. Here, V and Q cannot be factored further since they have no divisors. Thus, FACTOR(V) = (d|e) and FACTOR(Q) = (f|g). However, R can be factored more since {1,a} is a divisor. Thus, repeating the above steps on R, we obtain FACTOR(R) = (1|a)(b|c). Thus, the final factored form is (1|a)(b|c)|(d|e)(f|g).

(4.) Simplify final expression by eliminating "1". The term (1|a) in the final expression can be further simplified to a?. Thus, we obtain the desired factored form for S.

III. SELECTING

The step of selecting comprises selecting a DTD. In a preferred embodiment, the DTD comprises one or more sequences from the input sequences, generalized sequences, and factored sequences. Alternatively, the DTD may be selected from the input sequences and generalized sequences if a factoring step is not used. In a preferred embodiment the step of selecting is implemented using minimum descriptor length (MDL) principles.

The MDL cost of a DTD that is used to weigh a set of sequences, is comprised of:

(A) the length, in bits, needed to describe the DTD, and

(B) the length of the sequences, in bits, when encoded in terms of the DTD.

First, the number of bits required to describe the DTD is estimated (part (A) of the MDL cost). Let Σ be the set of subelement symbols that appear in sequences in I. Let M be the set of metacharacters |,* , +, ?, (,). Let the length of a DTD viewed as a string in $\Sigma \cup M$, be n. Then, the length of the DTD in bits is n $\log(|\Sigma| + |M|)$. As an example, let Σ consist of the elements a and b. The length in bits of the DTD a* b* is 4 * log(2 + 6) = 12. Similarly, the length in bits of the DTD (ab|abb)(aa|ab*) is 16 * 3 = 48.

<u>The Encoding Scheme</u> comprises the following steps:

(A) $seq(D, s) = \varepsilon$ if $D = s$. In this case, DTD D is a sequence of symbols from the alphabet Σ and does not contain any metacharacters.

(B) $seq(D_1 \ldots D_k, s_1 \ldots s_k) = (D_1, s_1) \ldots seq(D_k, s_k)$ that is, D is the concatenation of regular expressions $D_1 \ldots D_k$, and the sequence s can be written as the concatenation of the subsequences $s_1 \ldots s_k$, such that each subsequence s_i matches the corresponding regular expression D_i.

(C) $seq(D_1 | \ldots | D_m, s) = i\ seq(D_i, s)$ that is, D is the exclusive choice of regular expressions $D_1 \ldots D_m$, and i is the index of the regular expression that the sequence s matches. Note that we need $\lceil \log m \rceil$ bits to encode the index i.

(D) $seq(D^*, s_1 \ldots s_k) =$

In other words, the sequence $s = s_1 \ldots s_k$ is produced from D^* by instantiating the repetition operator k times, and each subsequence s_i matches the i-th instantiation. In this case, since there is no simple and inexpensive way to bound apriori, the number of bits required for the index k, we first specify the number of bits required to encode k in unary (that is, a sequence of $\lceil \log k \rceil$ 1s, followed by a 0) and then the index k using $\lceil \log k \rceil$ bits. The 0 in the middle serves as the delimiter between the unary encoding of the length of the index and actual index itself.

Table 4: Encoding Scheme

The MDL subsystem is responsible for choosing a set S of candidate DTDs from S_F such that the final DTD D (which is a logic OR of the DTDs in S) (1) encompasses all sequences in I, and (2) has the minimum MDL cost.

Next, the scheme for encoding a sequence using a DTD (part (B) of the MDL cost) is determined. The encoding scheme constructs a sequence of integral indices (which forms the encoding) for expressing a sequence in terms of a DTD. The following simple examples illustrate the basic building blocks on which the encoding scheme for more complex DTDs is built:

(1.) The encoding for the sequence a in terms of the DTD a is the empty string ϵ.
(2.) The encoding for the sequence b in terms of the DTD a | b | c is the integral index 1 (denotes that b is at position 1, counting from 0, in the above DTD).
(3.) The encoding for the sequence bbb in terms of the DTD b* is the integral index 3 (denotes 3 repetitions of b).

Next, the encoding scheme for arbitrary DTDs and arbitrary sequences is generalized. The sequence of integral indices for a sequence s when encoded is denoted in terms of a DTD D by seq(D,s). We define seq(D,s) recursively in terms of component DTDs within D as shown in Table 4. Thus, seq(D,s) can be computed using a recursive procedure based on the encoding scheme of the factoring algorithm depicted in Table 4. Note that the definitions of the encodings for operators + and ? have not been provided since these can be defined in a similar fashion to * (for +, k is always greater than 0, while for ?, k can only assume values 1 or 0).

Next the encoding scheme is illustrated using the following example. Consider the DTD (ab|c)* (de|f g*) and the sequence abccabfggg to be encoded in terms of the DTD. Below, we list how steps (A), (B), (C) and (D) in Table 4 are recursively applied to derive the encoding seq((ab|c)* (de|f g*); abccabf ggg).

1. Apply Step (B). seq((ab|c)* ; abccab))seq((de|f g*); f ggg)
2. Apply Step (D). 4 seq(ab|c, ab) seq(ab|c, c) seq(ab|c, c) seq(ab|c, ab) seq((de|f g*); f ggg)
3. Apply Step (C). 4 0 seq(ab, ab) 1 seq(c, c) 1 seq(c, c) 0 seq(ab, ab) 1 seq(f g*, f ggg)
4. Apply Step (A). 4 0 1 1 0 1 seq(f g*, f ggg)
5. Apply Steps (A), (B) and (D). 4 0 1 1 0 1 3

In order to derive the final bit sequence corresponding to the above indices, the unary representation for the number of bits required to encode the indices 4 and 3 is included in the encoding. Thus, the following bit encoding for the sequence is obtained:

seq((ab|c)* (de| fg*), abccabfggg) = 1110100 0 1 1 0 1 11011

In steps (B), (C) and (D), of the encoding scheme it needs to be determined if a sequence s matches a DTD D. Since a DTD is a regular expression, known techniques for finding out if a sequence is encompassed by a regular expression can be used. These known methods involve constructing a non-deterministic finite automaton for D and can also be used to decompose the sequence s into subsequences such that each subsequence matches the corresponding sub-part of the DTD D, thus enabling the encoding to be determined.

Note that there may be multiple ways of partitioning the sequence s such that each subsequence matches the corresponding sub-part of the DTD D. In such a case, the above procedure can be extended to enumerate every decomposition of s that match sub-parts of D, and then select from among the decompositions, the one that results in the minimum length encoding of s in terms of D.

Computing the DTD with Minimum MDL Cost

Next, the final DTD D (which is a logic OR of a subset S of candidate DTDs in S_F) that encompasses all the input sequences and whose MDL cost for encoding the input sequences is minimum is computed. The minimization problem maps naturally to the Facility

Location Problem (FLP). The Facility Location Problem is well known in the art. The FLP is formulated as follows: Let C be a set of customers and J be a set of facilities such that the facilities "serves" every customer. There is a cost $c(j)$ of "choosing" a facility $j \in J$ and a cost $d(j, i)$ of serving customers $i \in C$ by facility $j \in J$. The problem definition asks to choose a subset of facilities $F \subset J$ such that the sum of costs of the facilities plus the sum of costs of serving every client by its closest chosen facility is minimized, that is

The problem of inferring the minimum MDL cost DTD can be reduced to the FLP as follows: Let C be the set input sequences and J be the set of candidate DTDs in S_F . The cost of choosing a facility is the length of the corresponding candidate DTD. The cost of serving client i from facility j, $d(j, i)$, is the length of the encoding of the sequence corresponding to i using the DTD corresponding to the facility j. If a DTD j does not encompass a sequence i, then we set $d(j, i)$ to 1. Thus, the set F computed by the FLP corresponds to the desired set S of candidate DTDs. Algorithms for solving the FLP are well known in the art. In a preferred embodiment, a randomized algorithm is employed to approximate the FLP.

Having thus described a few particular embodiments of the invention, various alterations, modifications, and improvements will readily occur to those skilled in the art. Such alterations, modifications and improvements as are made obvious by this disclosure are intended to be part of this description though not expressly stated herein, and are intended to be within the spirit and scope of the invention. Accordingly, the foregoing description is by way of example only, and not limiting. The invention is limited only as defined in the following claims and equivalents thereto.

EXPERIMENTAL RESULTS

In order to determine the effectiveness of the present invention for inferring the DTD of a database of XML documents, we conducted a study with both synthetic and real-life DTDs. We also compared the DTDs produced by a DTD extraction tool (XTRACT) in accordance with a preferred embodiment of the present invention with those generated by the IBM alphaworks DTD extraction tool, DDbE (Data Description by Example), for XML data (the DDbE tool and a detailed description of it is available at http://www.alphaworks.ibm.com/). The results indicate that XTRACT outperforms DDbE over a wide range of DTDs, and accurately finds almost every original DTD while DDbE fails to do so for most DTDs. Thus, the results clearly demonstrate the effectiveness of XTRACT's approach that employs generalization and factorization to derive a range of general and concise candidate DTDs, and then uses the MDL principle as the basis to select from amongst them.

The two DTD extraction algorithms considered in the experimental study are as follows:

> XTRACT: XTRACT includes all three steps for determining a DTD in accordance with the present invention. In the generalization step, we discover both sequencing and OR patterns using procedure GENERALIZE. In the factoring step, $k = {}^{N}/_{10}$ subsets are chosen for factoring and the parameter ς is set to 0 in the procedure FACTORSUBSETS. Finally, in the selection step, we employ an algorithm which incorporate MDL principles to compute an approximation to the facility location problem (FLP).

> DDbE: We used Version 1.0 of the DDbE DTD extraction tool in the experiments. DDbE is a Java component library for inferring a DTD from a data set consisting of well-formed XML instances. DDbE offers parameters which permit the user to control the structure of the content models and the types used for attribute declarations. Some of the important parameters of DDbE that we used in the experiments, along with their default values, are presented in Table 5.

Parameter	Meaning	Default
c	Maximum number of consecutive identical tokens not replaced by a list	1
d	Maximum depth of factorization	2

Table 5: Description of Parameters Used by DDbE

The parameter c specifies the maximum number of consecutive identical tokens that should not be replaced by a list. For example, the default value of this parameter is 1 and thus all sequences containing two or more repetitions of the same symbol are replaced with a positive list. That is, aa is substituted by a+. The parameter d determines the number of applications of factoring. For a set of input sequences that conform to the DTD of a(b|c|d)(e|f|g)h, for increasing values of the parameter d, DDbE returns the DTDs in Table 6.

Parameter Value (d)	DTD Obtained
1	(acg|ace|adf|abg|abe|acf|adg|ade|abf)h
2	a(cg|ce|df|bg|be|cf|dg|de|bf)h
3	a((c|b|d)g|(d|c|b)f|(c|b|d)e)h
4	a((c|b|d)g|(d|c|b)f|(c|b|d)e)h

Table 6: DTDs generated by DDbE for Increasing Values of Parameter d

As shown in Table 6, for d = 1, factorization is performed once in which the rightmost symbol h is factored out. When the value of d becomes 2, the leftmost symbol a is also factored out. A further increase in the value of d to 3 causes factorization to be performed on

the middle portion of the expression and the common expression (b|c|d) to be extracted. However, note that subsequent increases in the value of d (beyond 3) do not result in further changes to the DTD. This seems to be a limitation of DDbE's factoring algorithm since examining the DTD for d = 3, we can easily notice that e, f and g have a common factor of (b|c|d) with different placement of the symbols within the parenthesis. However, the current version of DDbE cannot factorize this further.

In order to evaluate the quality of DTDs retrieved by XTRACT, we used both synthetic as well as real-life DTD schemas. For each DTD for a single element, we generated an XML file containing 1000 instantiations of the element. These 1000 instantiations were generated by randomly sampling from the DTD for the element. Thus, the initial set of input sequences I to both XTRACT and DDbE contained somewhere between 500 and 1000 sequences (after the elimination of duplicates) conforming to the original DTD.

THE DATA

Synthetic DTD Data Set: We used a synthetic data generator to generate the synthetic data sets. Each DTD is randomly chosen to have one of the following two forms: $A1|A2|A3|A_n$ and $A_1A_2A_3 \ldots A_n$. Thus, a DTD has n building blocks where n is a randomly chosen number between 1 and mb, where mb is an input parameter to the generator that specifies the maximum number of building blocks in a DTD. Each building block A_i further consists of n_i symbols, where n_i is randomly chosen to be between 1 and ms (the parameter ms specifies the maximum number of symbols that can be contained in a building block). Each building block A_i has one of the following four forms, each of which has an equal probability of occurrence: (1) $(a1|a2|a3| \ldots |a_{ni})$ (2) $a1a2a3 \ldots a_{ni}$ (3) $(a1|a2|a3|a4| \ldots |a_{ni})^*$ (4) $(a1a2a3a4 \ldots a_{ni})^*$. Here, the a_i's denote subelement symbols. Thus, the synthetic data

generator essentially generates DTDs containing one level of nesting of regular expression terms.

In Table 7, we show the synthetic DTDs that we considered in the experiments (note that, in Table 7, we only include the regular expression corresponding to the DTD). The DTDs were produced using the generator with the input parameters mb and ms both set to 5. Note that we use letters from the alphabet as subelement symbols.

<table>
<tr><th>No.</th><th>Original DTD</th></tr>
<tr><td>1</td><td>abcde|ef gh|ij|klm</td></tr>
<tr><td>2</td><td>(a|b|c|d|f)* gh</td></tr>
<tr><td>3</td><td>(a|b|c|d)* |e</td></tr>
<tr><td>4</td><td>(abcde)* f</td></tr>
<tr><td>5</td><td>(ab)* |cdef |(ghi)*</td></tr>
<tr><td>6</td><td>abcdef(g|h|i|j)(k|l|m|n|o)</td></tr>
<tr><td>7</td><td>(a|b|c)d* e* (f gh)*</td></tr>
<tr><td>8</td><td>(a|b)(cdefg)* hijklmnopq(r|s)*</td></tr>
<tr><td>9</td><td>(abcd)* |(e|f |g)* |h|(ijklm)*</td></tr>
<tr><td>10</td><td>a* |(b|c|d|e|f)* |gh|(i|j|k)* |(lmn)*</td></tr>
</table>

Table 7: Synthetic DTD Data Set

The ten synthetic DTDs vary in complexity with later DTDs being more complex than the earlier ones. For instance, DTD 1 does not contain any metacharacters, while DTDs 2 through 5 contain simple sequencing and OR patterns. DTD 6 represents a DTD in factored form while in DTDs 7 through 10, factors are combined with sequencing and OR patterns.

<u>Real-life DTD Data Set:</u> We obtained the real-life DTDs from the Newspaper Association of America (NAA) Classified Advertising Standards XML DTD produced by the NAA Classified Advertising Standards Task Force (this can be accessed at http://www.naa.org/technology/clsstdtf/Adex010.dtd). We examined this real-life DTD data and collected six representative DTDs that are shown in Table 8. Of the DTDs shown in the table, the last three DTDs are quite interesting. DTD 4 contains the metacharacter ? in conjunction with the metacharacter *, while DTDs 5 and 6 contain two regular expressions with * 's, one nested within the other.

<table>
<tr><th>No.</th><th>Original DTD</th><th>Simplified DTD</th></tr>
<tr><td>1</td><td><ENTITY % included-elements
"audio-clip|blind-box-reply|graphic|linkpi-char|video-clip"></td><td>a|b|c|d|e</td></tr>
<tr><td>2</td><td><ELEMENT communications-contacts
(phone|faxjemail|pager|web-page)*></td><td>(a|b|c|d|e)*</td></tr>
<tr><td>3</td><td><ELEMENT employment-services(employment-service.type;
employment-service.location * (e.zz-generic-tag)*)></td><td>ab* c*</td></tr>
<tr><td>4</td><td><ENTITY % location"addr* , geographic-area?, city?,
state-province?,postal-code?, country?"></td><td>a* b?c?d?</td></tr>
<tr><td>5</td><td><ELEMENT transfer-info(transfer-number; (from-to,
company-id)+,contact-info)*></td><td>(a(bc)+d)*</td></tr>
<tr><td>6</td><td><ELEMENT real-estate-services(real-estate-service.type,
real-estate-service.location?, r-e.response-modes*>
r-e.comment?)* ?</td><td>(ab?c* d?)*</td></tr>
</table>

Table 8: Real-life DTD Data Set

QUALITY OF INFERRED DTDS

Synthetic DTD Data Set: The DTDs inferred by XTRACT and DDbE for the synthetic data set are presented in Table 9. As shown in the table, XTRACT infers each of the original DTDs correctly. In contrast, DDbE computes the accurate DTD for only DTD 1 which is the simplest DTD containing no metacharacters. Even for the simple DTDs 2-5, not only is DDbE unable to correctly deduce the original DTD, but it also infers a DTD that does not encompass the set of input sequences. For instance, one of the input sequences encompassed by DTD 2 is gh which is not encompassed by the DTD inferred by DDbE. Thus, while XTRACT infers a DTD that encompasses all the input sequences, the DTD returned by DDbE may not encompass every input sequence. DTD 4 exemplifies the two typical behaviors of DDbE - (1) sequence f that is not frequently repeated is appended to both the front and the back of the final DTD, and (2) symbols that are repeated frequently are all OR'd together and encapsulated by the metacharacter +. For example, DDbE incorrectly identifies the term (abcde)* to be (a|b|c|d|e)* which is much more general. Thus, the DDbE tool has a tendency to over-generalize when the original DTDs contain regular expressions with * s. This same trend to over-generalize can be seen in DTDs 8-10 also. On the other hand, as is evident from Table 9, this is not the case for XTRACT which correctly infers every one of the original DTDs even for the more complex DTDs 8-10 that contain various combinations of sequencing and OR patterns. This clearly demonstrates the effectiveness of the generalization module in discovering these patterns and the MDL module in selecting these general candidate DTDs as the final DTDs.

Also, as discussed earlier, DDbE is not very good at factoring DTDs. For instance, unlike XTRACT, DDbE is unable to derive the final factored form for DTD 6. Finally,

DDbE infers an extremely complex DTD for the simple DTD 7. The results for the synthetic data set clearly demonstrate the superiority of XTRACT's approach (based on the combination of generalizing, factoring, and selecting using MDL principles) compared to DDbE's for the problem of inferring DTDs.

Real-life DTD Data Set: The DTDs generated by the two algorithms for the real-life data set are shown in Table 10. Of the five DTDs, XTRACT is able to infer all five correctly. In contrast, DDbE is able to derive accurate DTDs only for DTDs 1 and 2, and an approximate DTD for DTD 3. Basically, with an additional factoring step, DDbE could obtain the original DTD for DTD 3. Note, however, that DDbE is unable to infer the simple DTD 4 that contains the metacharacter ?. In contrast, XTRACT is able to deduce this DTD because its factorization step takes into account the identity element "1" and simplifies expressions of the form 1|a to a?. DTD 5 represents an interesting case where XTRACT is able to mine a DTD containing regular expressions containing nested * s. This is due to the generalization module that iteratively looks for sequencing patterns. On the other hand, DDbE simply over-generalizes the DTD 5 by ORing all the symbols in it and enclosing them within the metacharacter +.

No.	Original DTD	DTD Inferred by XTRACT	DTD Inferred by DDbE
1	abcde|ef gh|ij|klm	abcde|ef gh|ij|klm	abcde|efgh|ij|klm
2	(a|b|c|d|f)* gh	(a|b|c|d|f)* gh	gh(a|b|c|d|f)+gh
3	(a|b|c|d)* |e	(a|b|c|d)* |e	(e(a|c|d|b)+e)
4	(abcde)* f	(abcde)* f	(f (a|e|d|c|b)+f)
5	(ab)* |cdef|(ghi)*	(ab)* |cdef |(ghi)*	cdef(a|b|g|i|h)+cdef
6	abcdef(g|h|i|j)(k|l|m|n |o)	abcdef(g|h|i|j)(k|l|m|n|o)	abcdef(j(o|l|m|n|k)|g(o|l|n| m|k)|h(m|l|n|k|o)|i(o|l|n|m|

<table>
<tr><td></td><td></td><td></td><td>k))</td></tr>
<tr><td>7</td><td>(a|b|c)d* e* (f gh)*</td><td>(a|b|c)d* e* (f gh)*</td><td>((c|b|a)d+e+|ad+|bd+|c(e+|d+)?|ad*|be*))(f|h|g)+((a|b|c)d+e+|c(e+|d+)?|a(e+|d+)?|b(e+|d+)?)</td></tr>
<tr><td>8</td><td>(a|b)(cdef g)* hijklmnopq(r|s)*</td><td>(a|b)(cdefg)*hijklmnopq(r|s)*</td><td>((((a|b)hijabcdef g)|b|a)(c|g|f |e|d|s|r)+((b|a)?hijkamnop q))</td></tr>
<tr><td>9</td><td>(abcd)* |(e|f |g)* |h|(ijklm)*</td><td>(abcd)* |(i|klm)* |h|(e|f |g)*</td><td>h(a|d|c|b|e|g|f |i|m|l|k|j)+h</td></tr>
<tr><td>10</td><td>a* |(b|c|d|e|f)* |gh|(i|j|k)* | (lmn)*</td><td>a* |(b|c|d|e|f)* |gh|(i|j|k)* | (lmn)*</td><td>(a+|gh)(e|f |d|i|j|l|n|m|k|c|b)+(a+|gh)</td></tr>
</table>

Table 9: DTDs generated by XTRACT and DDbE for Synthetic Data Set

<table>
<tr><th>No.</th><th>Simplified DTD</th><th>DTD Obtained by XTRACT</th><th>DTD obtained by DDbE</th></tr>
<tr><td>1</td><td>a|b|c|d|e</td><td>a|b|c|d|e</td><td>a|b|c|d|e</td></tr>
<tr><td>2</td><td>(a|b|c|d|e)*</td><td>(a|b|c|d|e)*</td><td>(a|b|c|d|e)*</td></tr>
<tr><td>3</td><td>(ab* c*)</td><td>ab* c*</td><td>(ab+c*)|(ac*)</td></tr>
<tr><td>4</td><td>a* b?c?d?</td><td>a* b?c?d?</td><td>(a+b(c|(c?d))?)|((b|a+)?cd)|((a+|b)?d)|((a+|b)?c)|a+|b)</td></tr>
<tr><td>5</td><td>(a(bc)+d)*</td><td>(a(bc)* d)*</td><td>(a|b|c|d)+</td></tr>
</table>

Table 10: DTDs generated by XTRACT and DDbE for Real-life Data Set

The quality of the DTDs inferred by XTRACT was compared with those returned by the IBM alphaworks DDbE (Data Descriptors by Example) DTD extraction tool on synthetic as well as real-life DTDs. In the experiments, XTRACT outperformed DDbE by a wide margin, and for most DTDs it was able to accurately infer the DTD while DDbE completely failed to do so. A number of the DTDs which were correctly identified by XTRACT were

fairly complex and contained factors, metacharacters, and nested regular expression terms. Thus, the results clearly demonstrate the effectiveness of XTRACT's approach that employs generalization and factorization to derive a range of general and concise candidate DTDs, and then uses a selection step preferably comprising minimum descriptor length (MDL) principles as the basis to select from amongst them.

What is claimed is:

1. A document descriptor extraction method comprising the steps of:

generalizing input sequences associated with a document to develop general sequences, said input sequences reflecting the structure of a document;

factoring said input sequences and said general sequences to develop factored sequences;

selecting a document descriptor from said input sequences, said general sequences, and said factored sequences using minimum descriptor length (MDL) principles.

2. The method of claim 1, wherein said selecting step comprises the steps of:

encoding said input sequences, said general sequences, and said factored sequences; and

selecting a document descriptor which encompasses all of said input sequences and exhibits a minimum MDL cost.

3. The method of claim 2, wherein said encoding step comprising:

$seq(D,s) = \varepsilon$ if $D=s$, if D does not contain metacharacters;

$seq(D_1 \ldots D_k, s_1 \ldots s_k) = seq(D_1,s_1) \ldots seq(D_k,s_k)$;

$seq(D_1 | \ldots | D_m, s) = i\ seq(D_i,s)$;

$seq(D^*, s_1 \ldots s_k) = \{k\ seq(D,s_1) \ldots seq(D,s_k)$ if $k>0$; 0 otherwise$\}$;

wherein D is a sequence of symbols, s is a sequence, and i is an index of a regular expression that the corresponding sequence s matches, wherein log m bits are needed to encode index i.

4. The method of claim 3, wherein said minimum MDL cost is determined by employing an algorithm to solve a facility location problem (FLP), said FLP modified to compute said minimum MDL cost of potential document descriptors.

5. The method of claim 4, wherein said document descriptor is a document type descriptor (DTD), and said document is an eXtensible Markup Language (XML) document.

6. The method of claim 5, wherein said minimum MDL cost comprises summing a first length of bits describing the DTD and a second length of bits for encoding the sequences.

7. A document descriptor extraction method comprising the steps of:

generalizing input sequences to develop general sequences, said input sequences reflecting the structure of data within a document;

selecting a document descriptor from said input sequences and said general sequences using minimum descriptor length (MDL) principles.

8. The method of claim 7, wherein said selecting step comprises the steps of:

encoding said input sequences and said general sequences; and

selecting a document descriptor which encompasses all of said input sequences and exhibits a minimum MDL cost.

9. The method of claim 8, wherein said encoding step employs an algorithms which applies a set of rules comprising:

seq(D,s) = ε if D=s, if D does not contain metacharacters;

$seq(D_1...D_k, s_1...s_k) = seq(D_1,s_1)...seq(D_k,s_k)$, if D is a concatenation of $D_1...D_k$;

$seq(D_1|...|D_m,s) = i\ seq(D_i,s)$;

$seq(D^*,s_1...s_k) = \{k\ seq(D,s_1)...seq(D,s_k)$ if $k>0$; 0 otherwise$\}$;

wherein D is a sequence of symbols, s is a sequence, and i is an index of a regular expression that the corresponding sequence s matches, wherein log m bits are needed to encode index i.

10. The method of claim 9, wherein said minimum MDL cost is determined by employing an algorithm to solve a facility location problem (FLP), wherein said FLP is modified to compute said minimum MDL cost of potential document descriptors.

11. The method of claim 10, wherein said document descriptor is a document type descriptor (DTD), and said document is an eXtensible Markup Language (XML) document.

12. The method of claim 11, wherein said minimum MDL cost comprises summing a first length of bits describing the DTD and a second length of bits for encoding the sequences.

13. The method of claim 7, further comprising the step of:

factoring said input sequences and said general sequences to develop factored sequences, wherein said factored sequences are available for said step of selecting;

14. A computer program for generalizing input sequences to develop general sequences comprising:

a discover OR patterns procedure;

a discover sequence patterns procedure; and

a generalize procedure which calls said discover sequence patterns procedure and calls said discover OR patterns procedure, wherein said discover OR patterns procedure is nested within said discover sequence patterns procedure.

15. The computer program of claim 14, further comprising a partition procedure called by said discover OR patterns procedure.

16. A document descriptor extraction method of claim 15, utilizing a computer program for generalizing input sequences as set forth in claim 15.

17. The method of claim 16, comprising:

generalizing said input sequences to create general sequences using said computer program; and

selecting a document descriptor from said input sequences and said general sequences.

18. The method of claim 17, further comprising:

factoring said input sequences and said general sequences to develop factored sequences, wherein said factored sequences are available to said step of selecting.

19. The method of claim 18, wherein said step of selecting employs minimum descriptor length (MDL) principles.

20. The method of claim 19, wherein said document descriptor is a document type descriptor (DTD) and said document is an eXtensible Markup Language (XML) document.

21. A method for generalizing input sequences to develop general sequences comprising the steps of:

discovering OR patterns among said input sequences; and

discovering sequence patterns among said input sequences and OR patterns.

22. The method of claim 21, wherein said step of discovering OR patterns comprises the step of partitioning said input sequences.

23. A document descriptor extraction method, utilizing a method for generalizing input sequences as set forth in claim 22.

24. The method of claim 23, further comprising the steps of:

generalizing said input sequences to create general sequences using said method for generalizing input sequences; and

selecting a document descriptor from said input sequences and said general sequences.

25. The method of claim 24, further comprising the steps of:

factoring said input sequences and said general sequences to develop factored sequences, wherein said factored sequences are available to said step of selecting.

26. The method of claim 25, wherein said step of selecting employs minimum descriptor length (MDL) principles.

27. The method of claim 26, wherein said document descriptor is a document type descriptor (DTD) and said document is an eXtensible Markup Language (XML) document.

TITLE: DOCUMENT DESCRIPTOR EXTRACTION METHOD

ABSTRACT OF THE DISCLOSURE

The present invention discloses a document descriptor extraction method and system. The document descriptor extraction method and system creates a document descriptor by generalizing input sequences within a document; factoring the input sequences and generalized input sequences; and selecting a document descriptor from the input sequences, generalized sequences, and factored sequences, preferably using minimum descriptor length (MDL) principles. Novel algorithms are employed to perform the generalizing, factoring, and selecting.

M:\SWeed\Lucent Technologies\23,760\Patent Office\draft02.wpd

EXHIBIT C

Comparison of Filed Document
To April 28, 2000 Draft

Formatted: Indent: First line: 0", Line spacing: Exactly 12 pt

TITLE: DOCUMENT DESCRIPTOR EXTRACTION METHOD

FIELD OF THE INVENTION

The present invention relates to electronic documents. Specifically, the present invention relates to determining document descriptors from data within electronic documents.

BACKGROUND OF THE INVENTION

The number of documents available in electronic format has exploded. With the number of available electronic documents increasing rapidly, it is important to be able to quickly and accurately search the available electronic documents. In addition, it is desirable to be able to store data into electronic documents and generate new electronic documents which are similar in structure to existing electronic documents. Hence, tools which assist in the querying of electronic documents, the creation of electronic documents, and the storage of data into electronic documents are desirable.

Electronic documents for display over the Internet and/or an Intranet are commonly stored in a Standard Generalized Markup Language (SGML) format. SGML is a standard for how to specify a document markup language or tag set. SGML is not in itself a document language, but a description of how to specify one. The SGML format provides for the inclusion of a document type descriptor (DTD). A document's DTD specifies how the data within a document should be organized. One SGML format for storing data within electronic documents which is becoming increasingly popular is eXtensible Markup Language (XML). XML is rapidly emerging as the new standard for representing and exchanging data on the World Wide Web (web). An XML document may be accompanied by a document type

descriptor (DTD). For example, in an XML document, the DTD may specify the tags which can be used, the order in which the tags appear, how the tags are nested, and tag attributes. Thus, the DTD plays an important role in the storage of data to the XML document, the generation of similar documents, and increasing the efficiency of queries of the XML document. Efficiency is achieved by using the knowledge of the structure of the data to remove elements that cannot potentially satisfy the query.

Although DTDs are helpful in the storage, generation, and retrieval of data related to an XML document, DTDs are not mandatory. Since DTDs are not mandatory, many XML documents exist which do not contain DTDs. In addition, since only a small portion of the electronic documents in existence today are in an XML format, initially the majority of XML documents will likely be automatically generated from pre-existing non-XML documents. In many instances, the automatically generated XML formatted documents will not contain DTDs. Therefore, a tool for automatically generating DTDs is desirable for improving data storage and retrieval.

Others have attempted to automatically generate DTDs with varying degrees of success. One system is IBM's Data Descriptors by Example (DDbE) system. The goal of DDbE is to give users a good start at creating DTDs for their own applications. However, this system and other available systems do not produce highly accurate DTDs for all XML documents, especially complex XML documents. Since accurate DTDs enable efficient storage and retrieval of data, improved methods for extracting accurate DTDs from XML documents are desirable.

SUMMARY OF THE INVENTION

Deleted: Section Break (Continuous)

The present invention relates to developing a description of the layout of an electronic document from data within the document. The present invention is especially useful for determining document type descriptors (DTDs) of electronic documents in a Standard Generalized Markup Language (SGML) format.

The present invention comprises generalizing input sequences generated from an electronic document. The input sequence are generalized to create generalized sequences which are representative of the input sequences. Each generalized sequence encompasses one or more input sequences in a more general form. Next, the present invention comprises selecting a description of the layout of the electronic document from the input sequences and generalized sequences. Selecting a description comprises selecting one or more of the input sequences and generalized sequences such that every input sequence is encompassed by the selected sequences. Preferably, the selection is performed using minimum descriptor length (MDL) principles.

Additionally, the present invention may comprise factoring the input sequences and generalized sequences after generalizing to create factored sequences which can be included in the selection of the description. Each factored sequence encompasses one or more input sequences and generalized sequences. The factored sequence are combined with the input sequences and generalized sequences, thereby creating a potentially better selection of sequences from which a description may be selected.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is a flow chart of a preferred document type descriptor (DTD) extraction system in accordance with the present invention; and

Figure 2 is an illustrative depiction of the output of each step and the selection process of the preferred document type descriptor extraction system depicted in Figure 1 in accordance with the present invention.

Deleted: Section Break (Next Page)

DETAILED DESCRIPTION OF THE INVENTION

The present invention relates to inferring (i.e., determining) document descriptors from data within electronic documents. For illustrative purposes, the present invention is described in terms of inferring document type descriptors (DTDs) from data within eXtensible Markup Language (XML) formatted documents. However, it will be readily apparent to those skilled in the art that the present invention could be applied to other types of markup languages which provide document descriptions that are currently available or developed in the future, such as markup languages which conform to the Standard General Markup Language (SGML) format. The inferred DTD contains valuable information about the structure of the XML documents that it describes. The structural information may be used to efficiently query the XML document, store data to the XML document, or generate similar XML documents.

A sample XML document and its associated DTD are as follows:

Sample XML Document

```
<article>
  <title> A Relational Model </title>
```

Deleted: for Large Shared Data Banks

```
    <author>
        <name> A. B. Cod </name>
        <affiliation> ABC Corp. </affiliation>
    </author>
</article>
<article>
    <title> Another Model </title>
    <author>
        <name> A. Swift </name>
        <affiliation> XYZ Corp. </affiliation>
    </author>
    <author>
        <name> B. Quick </name>
        <affiliation> ZYX Corp. </affiliation>
    </author>
    <author>
        <name> C. Gold </name>
        <affiliation> XYZ Corp. </affiliation>
    </author>
    <author>
        <name> O. Henry </name>
        <affiliation> XYZ Corp. </affiliation>
    </author>
    <author>
        <name> E. Plant </name>
        <affiliation> XYZ Corp. </affiliation>
    </author>
</article>
```

Deleted: E. F. Codd

Deleted: IBM Research

Deleted: XTRACT: A system for Extracting DTDs

Deleted: M. Garofalakis

Deleted: Bell Labs

Deleted: A. Gionis

Deleted: Stanford University

Deleted: R. Rastogi

Deleted: Bell Labs

Deleted: Section Break (Continuous)

Deleted: S. Seshadri

Deleted: Bell Labs

Deleted: K. Shim

Deleted: Bell Labs

Sample Document Type Descriptor (DTD)

```
<!ELEMENT article (title, author*)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (name, affiliation)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT affiliation (#PCDATA)>
```

A DTD describes the structure of an XML document. A DTD constrains the structure of an element by specifying a regular expression with which its sub-element sequences must conform. The DTD declaration sequence uses commas for sequencing, | for exclusive OR,

parenthesis for grouping and meta-characters ?, *, + to denote zero or one, zero or more, and one or more, respectively.

In the sample XML document above and its associated DTD, the start of an element such as article is indicated by <article> and the end of the element is indicated by </article>. Each element may comprise sub-elements and/or data. For example, for the element article, title and author are sub-elements. Likewise, sub-elements may further contain additional sub-elements. For example, author contains sub-element name and sub-element affiliation.

In a preferred embodiment, the present invention applies algorithms in three steps to compute a DTD from a set of input sequences. They are (1) generalizing, (2) factoring, and (3) selecting.

Deleted: Section Break (Continuous)

The input sequences are groupings of sub-elements contained within each occurrence of an element. For an element, such as article, it is straight forward to compute the sequence of sub-elements nested within each <article> </article> pair in the XML document. The set of input sequences comprises one sequence for each occurrence of element <article>. For example, in the above XML document sample the input sequences for <article> would be input sequences <title><author> and <title><author><author><author><author><author>. For ease of description, the first letter of the sub-element may be used as a shorthand for describing sequences (e.g., <title> <author> is represented by ta and <title><author><author><author><author><author> is represented by taaaaa.)

In the preferred embodiment, the input sequences are generalized to create generalized sequences. The generalized sequences and input sequences are then factored to create factored sequences. Each factored sequence may encompass one or more input sequences and

generalized sequences, thereby creating additional sequences which may be selected as a part of a DTD. The factoring step is optional. However, using the factoring step results in potentially better DTDs. Factoring leads to better DTDs by creating additional sequences from which an appropriate DTD may be selected. A DTD which encompasses all of the input sequences is then selected from the input sequences, generalized sequences, and factored sequences.

Deleted: Section Break (Continuous)

In the generalization step, patterns within the input sequences are detected and more "general" regular expressions are substituted for them to create "generalized" sequences. In a preferred embodiment, the "generalized" sequences and the input sequences are then processed by the factorization step which factors common expressions to make them more succinct. The factorization step yields "factored" sequences. The first two steps along with the input sequences produce a series of potential DTDs that vary in their conciseness and precision. A selection step then selects a DTD from the candidates that strikes the right balance between conciseness and preciseness - that is, a DTD that is concise, but at the same time, is not too general. In a preferred embodiment, the selection step employs minimum descriptor length (MDL) principles for selecting a DTD.

Figure 1 depicts a flow chart 100 illustrating the steps for inferring a DTD in accordance with a preferred embodiment of the present invention. The input sequences I are comprised of sub-elements a, b, c, d, and e. The input sequences are first processed by a generalization module 110 which produces generalized sequences. The generalized sequences are combined with the input sequences to create a set of potential DTDs identified by S_G. Optionally, the potential DTDs are factored using a factoring module 120. The

factoring module produces additional potential DTDs which are combined with the potential DTDs output by the generalization module 110 to create a set of potential DTDs identified by S_F. Finally, the selecting module 130 infers (i.e. selects) a DTD from all of the potential DTDs S_F. Preferably, the selecting module 130 incorporates MDL principles.

Figure 2 graphically depicts the selection of a DTD from all of the potential DTDs. The selected DTD must encompass all of the original input sequences S_F. It can be seen that (ab)* encompasses input sequences ab and abab. Also, (a|b) (c|d) encompasses input sequences ac, ad, bc, and bd. Finally, b*(d|e) encompasses bd, bbd, and bbbbe. The selected potential DTDs, when combined using ORs, encompass all of the original input sequences. The result is a concise and precise DTD.

I. GENERALIZING

Deleted: Section Break (Continuous)

The quality of the data type descriptor (DTD) selected during the selection process is very dependent on the set of candidate DTDs available. If the selection were based on the input sequences only, then the final DTD output by the selection step would simply be the OR of all the input sequences. For example, in the above XML document sample, the DTD for <article> would comprise ta and taaaa (i.e., <title><author> and <title><author><author><author><author><author>.) However, this is not a desirable DTD since it is neither concise nor intuitive. A more concise and intuitive DTD would be the single sequence ta* which encompasses both ta and taaaaa. Thus, in order to infer meaningful DTDs, the candidate DTDs should be general. Ideally, each candidate DTD

encompasses more than one input sequence. The goal of the generalization module 110 is to achieve this objective.

The generalization module 110 of the present invention infers a number of regular expressions which have been found to frequently appear in real-life DTDs. Below, are examples of regular expressions from real-life DTDs that appear in the Newspaper Association of America (NAA) Classified Advertizing Standards XML DTD.

Deleted: (found at http://www.naa.org/technology/clsstdtf/Adex010.dtd)

a* bc* : DTDs of this form are generally used to specify tuples with set-valued attributes.
<!ELEMENT account-info (account-number, sub-account-number*)> <!-- Specification for account identification information -->

(abc)* : This type of DTD is used to represent a set (or a list) of ordered tuples.
<!ELEMENT days-and-hours (date, time)+> <!-- provide times/dates when job fairs will be held -->

(a|b|c)* : The DTD of the form (a|b|c)* is frequently used to represent a multiset containing the elements a, b and c. This DTD is very useful since the elements in the multiset are allowed to appear multiple times and in any order in the document. For example, the following DTD specifies that the support information for an ad can consist of an arbitrary number of audio or video clips, photos, and further these can appear in any order.
<!ELEMENT support-info (audio-clip | file-id | graphic | logo | new-list | photo | video-clip | zz-generic-tag)*> <!-- support information for ad content -->

((ab)* c)* : This type of DTD permits nesting relationships among sets (OR lists).
<!ELEMENT transfer-info (transfer-number, (from-to, company-id)+, contact-info)*> <!-- provides parent information through the multilevel aggregation process. may be repeated -->

Table 1 depicts pseudo code for a preferred generalization algorithm (Procedure GENERALIZE). Procedure GENERALIZE infers several DTDs for each input sequence independently and adds them to the set S_G. The generalize algorithm may over-generalize in some cases (since DTDs are inferred based on a single sequence), however, the selection step

in selecting module 130 will ensure that such overly-general DTDs are not chosen as part of the final inferred DTD, if there are better alternatives. The generalization step will provide several alternate candidates in addition to the input sequences for the selection step.

The algorithm can infer regular expressions that are more complex than the above, however, complex expressions, such as (ab?c* d?)*, that are less likely to occur in practice, may be excluded.

procedure GENERALIZE(I)
begin
1. **for each** sequence s in I
2. add s to S_g
3. **for** r := 2, 3, 4
4. s' := DISCOVERSEQPATTERN(s, r)
5. **for** $d := 0.1 \cdot |s'|, 0.5 \cdot |s'|, |s'|$
6. s'' := DISCOVERORPATTERN(s', d)
7. add s'' to S_g
end

procedure DISCOVERSEQPATTERN(s, r)
begin
1. **repeat**
2. let χ be a subsequence of s with the maximum number ($\geq r$) of contiguous repetitions in s
3. replace all ($\geq r$) contiguous occurrences of χ in s with a new auxiliary symbol $A_i = (\chi)^*$
4. **until** (s no longer contains $\geq r$ contiguous occurrences of any subsequence χ)
5. **return** s
end

procedure DISCOVERSEQPATTERN(s, d)
begin
1. $s_1, s_2, \ldots, s_n$:= PARTITION(s, d)
2. **for each** subsequence s_j in $s_1, s_2 \ldots, s_n$
3. let the set of distinct symbols in s_j be $a_1, a_2, \ldots, a_m$
4. **if** (m>1)
5. replace subsequence s_j in sequence s by a new auxiliary symbol $A_i = (a_1 | \cdots | a_m)^*$
6. **return** s
end

Deleted: Section Break (Continuous)
Formatted: Font: Not Italic
Formatted: Font: Not Italic

procedure PARTITION(s, d)
begin
1. $i := start := end := 1$
2. $s_i := s[start, end]$
3. **while** ($end < |s|$)
4.   **while** ($end < |s|$ **and** a symbol in s_i occurs to the right of s_i within a distance d)
5. $end := end + 1$; $s_i := s[start, end]$
6.   **if** ($end < |s|$)
7. $i := i + 1$; $start := end + 1$; $end := end + 1$; $s_i := s[start, end]$
8. **return** $s_1, s_2 \cdots, s_i$
end

Table 1: Generalization Algorithm

The essence of procedure GENERALIZE are the procedures DISCOVERSEQPATTERN and DISCOVERORPATTERN which are repeatedly called with predefined parameter values.

Discovering Sequencing Patterns (Procedure DISCOVERSEQPATTERN)

Procedure DISCOVERSEQPATTERN, shown in Table 1, takes an input sequence s and returns a candidate DTD that is derived from s by replacing sequencing patterns of the form xx... x, for a subsequence x in s, with the regular expression (x)*. In addition to s, the procedure also accepts as input, a threshold parameter r > 1 which is the minimum number of contiguous repetitions of subsequence x in s required for the repetitions to be replaced with (x)*. In case there are multiple subsequences x with the maximum number of repetitions in step 2 of procedure DISCOVERSEQPATTERNS, the longest among them is chosen, and subsequent ties are resolved arbitrarily.

Note that instead of introducing the regular expression term (x)* into the sequence s, an auxiliary symbol that serves as a representative for the term is introduced. The use of

auxiliary symbols enable the description of the algorithms to remain simple and clean since the input to them is always a sequence of symbols. In a preferred embodiment, there is a one-to-one correspondence between auxiliary symbols and regular expression terms in the present invention; thus, if the auxilliary symbol A denotes (bc)* in one candidate DTD, then it represents (bc)* in every other candidate DTD. Also, procedure DISCOVERSEQPATTERN may perform several iterations and thus new sequencing patterns may contain auxiliary symbols corresponding to patterns replaced in previous iterations. For example, invoking procedure DISCOVERSEQPATTERN with the input sequence s = abababcababc and r = 2 yields the sequence A_1cA_1c after the first iteration, where A_1 is an auxiliary symbol for the term (ab)*. After the second iteration, the procedure returns the candidate DTD A_2, where A_2 is the auxiliary symbol corresponding to ((ab)* c)*. Thus, the resulting candidate DTD returned by procedure DISCOVERSEQPATTERN can contain *s nested within other *s. Finally, DISCOVERSEQPATTERN is invoked with three different values for the parameter r to control the aggressiveness of the generalization. For example, for the sequence aabbb, DISCOVERSEQPATTERN with r = 2 would infer a* b*, while with r = 3, it would infer aab*. In the selection step, if many other sequences are encompassed by aab*, then a DTD of aab* may be preferred to a DTD of a* b* since it more accurately describes the input sequences.

Discovering OR Patterns (Procedure DISCOVERORPATTERN)

Procedure DISCOVERORPATTERN, shown in Table 1, infers patterns of the form $(a_1|a_2| \ldots |a_m)^*$ based on the locality of these symbols within a sequence s. The locality is

identified by first partitioning (performed by procedure PARTITION, shown in Table 1) the input sequence s into the smallest possible subsequences $s_1, s_2, \ldots, s_n$, such that for any occurrence of a symbol a in a subsequence s_i, there does not exist another occurrence of a in some other subsequence s_j within a distance d (which is a parameter to DISCOVERORPATTERN). Each subsequence s_i in s is then replaced by the pattern $(a_1|a_2| \ldots |a_m)^*$ where $a_1, \ldots, a_m$ are the distinct symbols in the subsequence s_i. If s_i contains frequent repetitions of the symbols $a_1|a_2|\ldots|a_m$ in close proximity, then it is very likely that s_i originated from a regular expression of the form $(a_1|a_2|\ldots|a_m)^*$. For illustrative purposes, for the input sequence abcbac, procedure DISCOVERORPATTERN returns:

- aA_1ac for $d = 2$, where $A_1 = (b \mid c)^*$;
- aA_2 for $d = 3$, where $A_2 = (a \mid b \mid c)^*$; and
- A_2 for $d = 4$, where $A_2 = (a \mid b \mid c)^*$.

A preferred component for discovering OR patterns is procedure PARTITION, shown in Table 1. For a sequence s, s[i,j] denotes the subsequence of s starting at the i^{th} symbol and ending at the j^{th} symbol of s. Procedure PARTITION constructs the subsequences in the order s_1, s_2, and so on. Assuming that s_1 through s_j have been generated, it constructs s_{j+1} by starting s_{j+1} immediately after s_j ends and expanding the subsequence s_{j+1} to the right as long as there is a symbol in s_{j+1} that occurs within a distance d to the right of s_{j+1}. By construction, there cannot exist such a symbol to the left of s_{j+1}.

Deleted: required to ensure that

Deleted: no

Note that procedure GENERALIZE invokes DISCOVERORPATTERN on the DTDs that result from calls to DISCOVERSEQPATTERN and therefore it is possible to infer more complex DTDs of the form $(a|(bc)^*)^*$ in addition to DTDs like $(a|b|c)^*$. For instance, for the

input sequence s = abcbca, procedure DISCOVERSEQPATTERN invoked with r = 2 would return s' = aA_1a, where $A_1 = (bc)^*$, which, when input to DISCOVERORPATTERN returns s'' = A_2 for d = |s'|, where $A_2 = (a|A_1)^*$. Further, DISCOVERORPATTERN is invoked with various values of d (expressed as a fraction of the length of the input sequence) to control the degree of generalization. Small values of d lead to conservative generalizations while larger values result in more liberal generalizations. The size of d is based on desired design characteristics.

II. FACTORING

In a preferred embodiment, the factoring module 120 uses a factoring step to derive factored forms for expressions that are an OR of a subset of the candidate DTDs, S_G, out of the generalization module 110. For example, for candidate DTDs ac, ad, bc and bd in S_G, the factoring step would generate the factored form (a | b)(c | d). Note that since the final DTD is an OR of candidate DTDs, S_F, out of the factoring module 120, the factored forms are also candidates. Further, a factored candidate DTD, because of its smaller size, has a lower minimum description length (MDL) cost, and is thus more likely to be chosen in the selection step, if MDL principles are used. Thus, since factored forms (due to their compactness) are more desirable, factoring can result in better quality DTDs.

Factored DTDs are common in real life. For example, in the sample DTD, an article may be categorized based on whether it appeared in a workshop, conference or journal; it may also be classified according to its area as belonging to either computer science, physics, chemistry etc. Thus, the DTD (in factored form) for the element article would be as follows:

<!ELEMENT article(title, author*, (workshop | conference | journal), (computer science | physics | chemistry | ...))

The set of candidate DTDs, S_F, output by the factorization module, 120, in addition to the factored forms generated from candidates in S_G, also contains all the DTDs in S_G. Ideally, factored forms for every subset of S_G, should be added to S_F to be considered by the selection step. However, this may be impractical, since S_G could be very large. Therefore, a heuristic may be used to select subsets of candidates in S_G that when factored yield good factored DTDs. In a preferred embodiment, the factoring algorithm is an adaptation of factoring algorithms for boolean expressions which are well known in the art.

<u>Selecting Subsets of S_G to Factor</u>

Intuitively, a subset S of S_G out of generalization module 110 is a good candidate for factoring if the factored form of S is much smaller than S itself. In addition, even though S_G may contain multiple generalizations that are derived from the same input sequence, it is highly unlikely that the final DTD will contain two generalizations of the same input sequence. Thus, factoring candidate DTDs in S_G that encompass similar sets of input sequences does not lead to factors that can improve the quality of the final DTD.

For a subset S of S_G to yield good factored forms it must satisfy the following two properties:

(1.) Every DTD in S has a common prefix or suffix with a number of other DTDs in S. Further, as more DTDs in S share common prefixes or suffixes, or as the length of the common prefixes/suffixes increases, the quality of the generated factored form can be expected to improve.

(2.) The overlap between every pair of DTDs D; D' in S is minimal, that is, the intersection of the input sequences encompassed by D and D' is small. This is important because, as mentioned above, a factored DTD adds little value (from an MDL cost perspective) over the candidate DTDs from which it was derived if it cannot be used to encode a significantly larger number of input sequences compared to the sequences encompassed by each individual DTD.

Deleted: . The

In order to state properties (1) and (2) for a set S of DTDs more formally, the following notation is used. For a DTD D, let cover(D) denote the input sequences in I that are encompassed by D (note that auxiliary symbols are expanded completely when cover for a DTD is computed). Then, overlap(D, D') is defined as the fraction of the input sequences encompassed by D and D' that are common to D and D', that is,

Deleted: Section Break (Continuous)

(1)

Thus, for a sufficiently small value of a (user-specified) parameter δ, by ensuring that overlap(D,D') < δ for every pair of DTDs D and D' in S, it can be ensured that S satisfies property (2) mentioned above.

In order to characterize property (1) more rigorously, the function score(D,S) is introduced in equation 2. Function score (D, S) attempts to capture the degree of similarity between prefixes/suffixes of DTD D and those of DTDs in the set S of DTDs. Intuitively, a DTD with a high score with respect to set S is a good candidate to be factored with other DTDs in set S. For a DTD D, let pref (D) and suf(D) denote the set of prefixes and suffixes of D, respectively. Let psup(p,S) denote the support of prefix p in set S of DTDs, that is, the number of DTDs in S for which p is a prefix. Similarly, let ssup(s,S) denote number of DTDs in S for which s is a suffix. Then score(D,S) is defined as follows:

$$\text{score}(D,S) = \max(\{|p| \cdot \text{psup}(p,S) : p \in \text{pref}(D)\} \cup \{|s| * \text{ssup}(s,S) : s \in \text{suf}(D)\}) \quad (2)$$

Thus, the prefix/suffix p=s of D, for which the product of p=s's length and its support in S is maximum, determines the score of D with respect to S. If DTD D has a long prefix or suffix that occurs frequently in set S, then this prefix can be factored out, thus resulting in good factored forms. The function score is thus a good measure of how well D would factor with other DTDs in S.

Procedure FACTORSUBSETS, shown in Table 2, first selects subsets S of sequences from within sequences S_G that satisfy properties (1) and (2). Each of these subsets S is then factored by invoking procedure FACTOR (in Step 15), depicted in Table 3. Assuming that the factoring algorithm returns $F_1 \mid F_2 \mid \ldots F_m$, each of the F_i is added to S_F.

procedure FACTORSUBSETS(S_g)
begin
1. **for each** DTD D is S_g — Deleted: Section Break (Continuous)
2. Compute *score* (D,S_g)
3. $S_F := S' := S_g$; SeedSet := $\emptyset$
4. **for** $i := 1$ to k
5. let D be the DTD in S' with the maximum value for *score* (D,S_g)
6. SeedSet := SeedSet $\cup$ D
7. $S' := S' - \{D' : overlap\,(D, D') \geq \delta\}$
8. **for each** DTD D in SeedSet
9. $S := \{D\}$
10. $S' := S_g - \{D' : overlap\,(D, D') \geq \delta\}$
11. **while** (S' is not empty)
12. let D' be the DTD in S' with the maximum value for *score* (D',S)
13. $S := S \cup D'$
14. $S' := S' - \{D' : overlap\,(D', D'') \geq \delta\}$
15. F := FACTOR(S)
16. $S_F := S_F \cup \{F_1;\ldots,F_m\}$ /* $F = F_1|\cdots|F_m$ */

end

Table 2: Choosing Subsets Of S_g For Factoring

Procedure FACTORSUBSETS computes a set S of candidate DTDs to factor. First, k seed DTDs for the sets S to be factored are chosen in the for loop spanning steps 4-7. These seed DTDs have a high score value with respect to S_G and overlap minimally with each other. Thus, it is ensured that each seed DTD not only factors well with other DTDs in S_G, but is also significantly different from other seeds. In steps 9-14, each seed DTD is used to construct a new set S of DTDs to be factored (thus, only k sets of DTDs are generated). After initializing S to a seed DTD D, in each subsequent iteration, the next DTD D' that is added to S is chosen greedily (i.e., the one whose score with respect to DTDs in S is maximum and whose overlap with DTDs already in S is less than δ).

Algorithm For Factoring a Set of DTDs

Deleted: ¶
¶

Deleted:

Deleted: Section Break (Continuous)

Algorithms for computing the optimum factored form, that is, the one with the minimum number of literals are known in the art. However, the complexity of these known techniques may be impractical. In a preferred embodiment, heuristic factoring algorithms for boolean functions which are known in the art are adapted for use in the present invention. Factored forms of boolean functions are commonly used in VLSI design.

There is a close correspondence between the semantics of DTDs and those of boolean expressions. The sequencing operator (,) in DTDs is similar to a logical AND in boolean algebra, while the OR operator (|) is like a logical OR. However, there exist certain fundamental differences between DTDs and boolean expressions. First, while the logical AND operator in boolean logic is commutative, the sequencing operator in DTDs is not (the ordering of symbols in a sequence matters!). Second, in boolean logic, the expression a | ab

is equivalent to a; however, the equivalent DTD for a | ab is ab?. The boolean algorithms can be modified to create a factoring algorithm to handle the semantics of the DTDs. The pseudo-code for procedure FACTOR, is shown in Table 3. Procedure FACTOR is a preferred embodiment of the factoring algorithm used in factoring module 120.

```
procedure FACTOR(S)        /* S is the set of sequences to be factored */
begin
DivisorSet := FINDALLDIVISORS(S)
if (DivisorSet = ∅)
    return or of sequences in S
DivisorList := ∅
for each divisor V in DivisorSet
    Q, R := DIVIDE(S, V)
    add (V, Q, R) to DivisorList
find the most compact triplet (V_i Q_i R_i) in DivisorList
return (FACTOR(V_i))(FACTOR(Q_i)) | FACTOR(R_i)
end

procedure FINDALLDIVISORS(S)
begin
DivisorSet := ∅
for each distinct sequence s such that s is a suffix for at least two elements in S
    DivisorSet := DivisorSet ∪ {{p : ps ε S}}
return DivisorSet
end

procedure DIVIDE(S, V)
begin
for each sequence p and V
    q_p := {s : ps ε S}
Q := ∩_{pεV} q_p
R := S - V ∘ Q
/* V ∘ Q is the set of sequences resulting from concatenating
    every sequence in Q to the end of every sequence in V */
return Q, R
end
```

Deleted: Section Break (Continuous).

Table 3: Factoring Algorithm

As an example of the factoring algorithm, consider the set S = {b, c, ab, ac, df, dg, ef, eg} of input sequences corresponding to the expression b|c|ab|ac|df|dg|ef|eg whose factored form is a?(b|c)|(d|e)(f|g). Before the steps that procedure FACTOR performs to derive the factored form are discussed, the DIVIDE operation that constitutes the core of the factoring algorithm is introduced. For sets of sequences S, V, DIVIDE(S,V) returns a quotient Q and remainder V such that $S = V \circ Q \cup R$ (here, $V \circ Q$ is the set of sequences resulting from concatenating every sequence in Q to the end of every sequence in V). Thus, for the above set S and V = {d,e}, DIVIDE(S,V) returns the quotient Q = {f,g} and remainder R = {b,c,ab,ac}. The steps executed by FACTOR to generate the factored form are as follows:

(1.) Compute set of potential divisors for S. These are simply sets of prefixes that have a common suffix in S. Thus, potential divisors for S include {d, e} (both f and g are common suffixes) and {1,a} (both b and c are common suffixes). The symbol "1" is special and denotes the identity symbol with respect to the sequencing operator, that is, 1s = s1 = s for every sequence s.

(2.) Choose divisor V from set of potential divisors. This is carried out by first dividing S by each potential divisor V to obtain a quotient Q and remainder R, and then selecting the V for which the triplet (V,Q,R) has the smallest size. In our case, V = {d,e} results in a smaller quotient and remainder (Q = {f, g}, R = {b, c, ab, ac}) than {1,a} (Q = {b,c}, R = {df,dg,ef,eg}) and is thus chosen.

(3.) Recursively factor V, Q, and R. The final factored form is FACTOR(V)FACTOR(Q)|FACTOR(R), where V = {d|e}, Q = {f,g} and R = {b,c,ab,ac}. Here, V and Q cannot be factored further since they have no divisors. Thus, FACTOR(V) = (d|e) and FACTOR(Q) = (f|g). However, R can be factored more since {1,a} is a divisor. Thus, repeating the above steps on R, we obtain FACTOR(R) = (1|a)(b|c). Thus, the final factored form is (1|a)(b|c)|(d|e)(f|g).

Deleted: Section Break (Continuous)

(4.) Simplify final expression by eliminating "1". The term (1|a) in the final expression can be further simplified to a?. Thus, we obtain the desired factored form for S.

III. SELECTING

The step of selecting comprises selecting a DTD. In a preferred embodiment, the DTD comprises one or more sequences from the input sequences, generalized sequences, and factored sequences. Alternatively, the DTD may be selected from the input sequences and generalized sequences if a factoring step is not used. In a preferred embodiment the step of selecting is implemented using minimum descriptor length (MDL) principles.

Deleted: ¶

The MDL cost of a DTD that is used to weigh a set of sequences, is comprised of:

(A) the length, in bits, needed to describe the DTD, and

(B) the length of the sequences, in bits, when encoded in terms of the DTD.

Deleted: ¶

First, the number of bits required to describe the DTD is estimated (part (A) of the MDL cost). Let Σ be the set of subelement symbols that appear in sequences in I. Let M be the set of metacharacters |,* , +, ?, (,). Let the length of a DTD viewed as a string in $\Sigma \cup M$, be n. Then, the length of the DTD in bits is $n \log(|\Sigma| + |M|)$. As an example, let Σ consist of the elements a and b. The length in bits of the DTD a* b* is $4 * \log(2 + 6) = 12$. Similarly, the length in bits of the DTD (ab|abb)(aa|ab*) is $16 * 3 = 48$.

The Encoding Scheme comprises the following steps:

(A) $seq(D, s) = \varepsilon$ if $D = s$. In this case, DTD D is a sequence of symbols from the alphabet Σ and does not contain any metacharacters.

Deleted: Section Break (Continuous)

(B) $seq(D_1 \ldots D_k, s_1 \ldots s_k) = (D_1, s_1) \ldots seq(D_k, s_k)$ that is, D is the concatenation of regular expressions $D_1 \ldots D_k$, and the sequence s can be written as the concatenation of the subsequences $s_1 \ldots s_k$, such that each subsequence s_i matches the corresponding regular expression D_i.

(C) $seq(D_1 | \ldots | D_m, s) = i\ seq(D_i, s)$ that is, D is the exclusive choice of regular expressions $D_1 \ldots D_m$, and i is the index of the regular expression that the sequence s matches. Note that we need $\lceil \log m \rceil$ bits to encode the index i.

(D) $seq(D^*, s_1...s_k) =$

In other words, the sequence $s = s_1...s_k$ is produced from D^* by instantiating the repetition operator k times, and each subsequence s_i matches the i-th instantiation. In this case, since there is no simple and inexpensive way to bound apriori, the number of bits required for the index k, we first specify the number of bits required to encode k in unary (that is, a sequence of $\lceil \log k \rceil$ 1s, followed by a 0) and then the index k using $\lceil \log k \rceil$ bits. The 0 in the middle serves as the delimiter between the unary encoding of the length of the index and actual index itself.

Table 4: Encoding Scheme

The MDL subsystem is responsible for choosing a set S of candidate DTDs from S_F such that the final DTD D (which is a logic OR of the DTDs in S) (1) encompasses all sequences in I, and (2) has the minimum MDL cost.

Next, the scheme for encoding a sequence using a DTD (part (B) of the MDL cost) is determined. The encoding scheme constructs a sequence of integral indices (which forms the encoding) for expressing a sequence in terms of a DTD. The following simple examples illustrate the basic building blocks on which the encoding scheme for more complex DTDs is built:

(1.) The encoding for the sequence a in terms of the DTD a is the empty string ϵ.
(2.) The encoding for the sequence b in terms of the DTD a | b | c is the integral index 1 (denotes that b is at position 1, counting from 0, in the above DTD).
(3.) The encoding for the sequence bbb in terms of the DTD b* is the integral index 3 (denotes 3 repetitions of b).

Next, the encoding scheme for arbitrary DTDs and arbitrary sequences is generalized.

Deleted: Section Break (Continuous)

The sequence of integral indices for a sequence s when encoded is denoted in terms of a DTD D by seq(D,s). We define seq(D,s) recursively in terms of component DTDs within D as

shown in Table 4. Thus, seq(D,s) can be computed using a recursive procedure based on the encoding scheme of the factoring algorithm depicted in Table 4. Note that the definitions of the encodings for operators + and ? have not been provided since these can be defined in a similar fashion to * (for +, k is always greater than 0, while for ?, k can only assume values 1 or 0).

Next the encoding scheme is illustrated using the following example. Consider the DTD (ab|c)* (de|f g*) and the sequence abccabfggg to be encoded in terms of the DTD. Below, we list how steps (A), (B), (C) and (D) in Table 4 are recursively applied to derive the encoding seq((ab|c)* (de|f g*); abccabf ggg).

1. Apply Step (B). seq((ab|c)* ; abccab))seq((de|f g*); f ggg)
2. Apply Step (D). 4 seq(ab|c, ab) seq(ab|c, c) seq(ab|c, c) seq(ab|c, ab) seq((de|f g*); f ggg)
3. Apply Step (C). 4 0 seq(ab, ab) 1 seq(c, c) 1 seq(c, c) 0 seq(ab, ab) 1 seq(f g*, f ggg)
4. Apply Step (A). 4 0 1 1 0 1 seq(f g*, f ggg)
5. Apply Steps (A), (B) and (D). 4 0 1 1 0 1 3

In order to derive the final bit sequence corresponding to the above indices, the unary representation for the number of bits required to encode the indices 4 and 3 is included in the encoding. Thus, the following bit encoding for the sequence is obtained:

seq((ab|c)* (de| fg*), abccabfggg) = 1110100 0 1 1 0 1 11011

Deleted: Section Break (Continuous)

In steps (B), (C) and (D), of the encoding scheme it needs to be determined if a sequence s matches a DTD D. Since a DTD is a regular expression, known techniques for finding out if a sequence is encompassed by a regular expression can be used. These known methods involve constructing a non-deterministic finite automaton for D and can also be used

to decompose the sequence s into subsequences such that each subsequence matches the corresponding sub-part of the DTD D, thus enabling the encoding to be determined.

. Note that there may be multiple ways of partitioning the sequence s such that each subsequence matches the corresponding sub-part of the DTD D. In such a case, the above procedure can be extended to enumerate every decomposition of s that match sub-parts of D, and then select from among the decompositions, the one that results in the minimum length encoding of s in terms of D.

Computing the DTD with Minimum MDL Cost

Next, the final DTD D (which is a logic OR of a subset S of candidate DTDs in S_F) that encompasses all the input sequences and whose MDL cost for encoding the input sequences is minimum is computed. The minimization problem maps naturally to the Facility Location Problem (FLP). The Facility Location Problem is well known in the art. The FLP is formulated as follows: Let C be a set of customers and J be a set of facilities such that the facilities "serves" every customer. There is a cost c(j) of "choosing" a facility $j \varepsilon J$ and a cost d(j, i) of serving customers $i \varepsilon C$ by facility $j \varepsilon J$. The problem definition asks to choose a subset of facilities $F \subset J$ such that the sum of costs of the facilities plus the sum of costs of serving every client by its closest chosen facility is minimized, that is

The problem of inferring the minimum MDL cost DTD can be reduced to the FLP as follows: Let C be the set input sequences and J be the set of candidate DTDs in S_F . The cost

of choosing a facility is the length of the corresponding candidate DTD. The cost of serving client i from facility j, d(j, i), is the length of the encoding of the sequence corresponding to i using the DTD corresponding to the facility j. If a DTD j does not encompass a sequence i, then we set d(j, i) to 1. Thus, the set F computed by the FLP corresponds to the desired set S of candidate DTDs. Algorithms for solving the FLP are well known in the art. In a preferred embodiment, a randomized algorithm is employed to approximate the FLP.

Having thus described a few particular embodiments of the invention, various alterations, modifications, and improvements will readily occur to those skilled in the art. For example, the invention may be embodied in computer program instructions stored in a computer-readable medium, e.g., floppy disc, hard drive, CD ROM, DVD, ROM, RAM, punch card, magnetic tape, etc. Such alterations, modifications and improvements as are made obvious by this disclosure are intended to be part of this description though not expressly stated herein, and are intended to be within the spirit and scope of the invention. Accordingly, the foregoing description is by way of example only, and not limiting. The invention is limited only as defined in the following claims and equivalents thereto.

Deleted: Section Break (Continuous)

EXPERIMENTAL RESULTS

In order to determine the effectiveness of the present invention for inferring the DTD of a database of XML documents, we conducted a study with both synthetic and real-life DTDs. We also compared the DTDs produced by a DTD extraction tool (XTRACT) in accordance with a preferred embodiment of the present invention with those generated by the IBM alphaworks DTD extraction tool, DDbE (Data Description by Example). The results

Deleted:), for XML data (the DDbE tool and a detailed description of it is available at http://www.alphaworks.ibm.com/

Formatted: Default Paragraph Font

Formatted: No underline

indicate that XTRACT outperforms DDbE over a wide range of DTDs, and accurately finds almost every original DTD while DDbE fails to do so for most DTDs. Thus, the results clearly demonstrate the effectiveness of XTRACT's approach that employs generalization and factorization to derive a range of general and concise candidate DTDs, and then uses the MDL principle as the basis to select from amongst them.

The two DTD extraction algorithms considered in the experimental study are as follows:

XTRACT: XTRACT includes all three steps for determining a DTD in accordance with the present invention. In the generalization step, we discover both sequencing and OR patterns using procedure GENERALIZE. In the factoring step, $k = {}^{N}/_{10}$ subsets are chosen for factoring and the parameter ς is set to 0 in the procedure FACTORSUBSETS. Finally, in the selection step, we employ an algorithm which incorporate MDL principles to compute an approximation to the facility location problem (FLP).

DDbE: We used Version 1.0 of the DDbE DTD extraction tool in the experiments. DDbE is a Java component library for inferring a DTD from a data set consisting of well-formed XML instances. DDbE offers parameters which permit the user to control the structure of the content models and the types used for attribute declarations. Some of the important parameters of DDbE that we used in the experiments, along with their default values, are presented in Table 5.

Deleted: Section Break (Continuous)
Formatted Table

Parameter	Meaning	Default
c	Maximum number of consecutive identical tokens not replaced by a list	1
d	Maximum depth of factorization	2

Table 5: Description of Parameters Used by DDbE

The parameter c specifies the maximum number of consecutive identical tokens that should not be replaced by a list. For example, the default value of this parameter is 1 and thus

all sequences containing two or more repetitions of the same symbol are replaced with a positive list. That is, aa is substituted by a+. The parameter d determines the number of applications of factoring. For a set of input sequences that conform to the DTD of a(b|c|d)(e|f|g)h, for increasing values of the parameter d, DDbE returns the DTDs in Table 6.

Parameter Value (d)	DTD Obtained
1	(acg|ace|adf|abg|abe|acf|adg|ade|abf)h
2	a(cg|ce|df|bg|be|cf|dg|de|bf)h
3	a((c|b|d)g|(d|c|b)f|(c|b|d)e)h
4	a((c|b|d)g|(d|c|b)f|(c|b|d)e)h

Formatted Table

Formatted: Centered

Table 6: DTDs generated by DDbE for Increasing Values of Parameter d

Deleted: Section Break (Continuous)

As shown in Table 6, for d = 1, factorization is performed once in which the rightmost symbol h is factored out. When the value of d becomes 2, the leftmost symbol a is also factored out. A further increase in the value of d to 3 causes factorization to be performed on the middle portion of the expression and the common expression (b|c|d) to be extracted. However, note that subsequent increases in the value of d (beyond 3) do not result in further changes to the DTD. This seems to be a limitation of DDbE's factoring algorithm since examining the DTD for d = 3, we can easily notice that e, f and g have a common factor of (b|c|d) with different placement of the symbols within the parenthesis. However, the current version of DDbE cannot factorize this further.

In order to evaluate the quality of DTDs retrieved by XTRACT, we used both synthetic as well as real-life DTD schemas. For each DTD for a single element, we generated an XML file containing 1000 instantiations of the element. These 1000 instantiations were generated by randomly sampling from the DTD for the element. Thus, the initial set of input sequences I to both XTRACT and DDbE contained somewhere between 500 and 1000 sequences (after the elimination of duplicates) conforming to the original DTD.

Deleted: ¶

THE DATA

Synthetic DTD Data Set: We used a synthetic data generator to generate the synthetic data sets. Each DTD is randomly chosen to have one of the following two forms: $A_1 | A_2 | A_3 | A_n$ and $A_1A_2A_3 \ldots A_n$. Thus, a DTD has n building blocks where n is a randomly chosen number between 1 and mb, where mb is an input parameter to the generator that specifies the maximum number of building blocks in a DTD. Each building block A_i further consists of n_i symbols, where n_i is randomly chosen to be between 1 and ms (the parameter ms specifies the maximum number of symbols that can be contained in a building block). Each building block A_i has one of the following four forms, each of which has an equal probability of occurrence: (1) $(a_1|a_2|a_3| \ldots |a_{ni})$ (2) $a_1a_2a_3 \ldots a_{ni}$ (3) $(a_1|a_2|a_3|a_4| \ldots |a_{ni})^*$ (4) $(a_1a_2a_3a_4 \ldots a_{ni})^*$. Here, the a_i's denote subelement symbols. Thus, the synthetic data generator essentially generates DTDs containing one level of nesting of regular expression terms.

Formatted: Subscript
Formatted: Subscript
Formatted: Subscript

In Table 7, we show the synthetic DTDs that we considered in the experiments (note that, in Table 7, we only include the regular expression corresponding to the DTD). The

Formatted: Subscript
Formatted: Subscript
Formatted: Subscript
Formatted: Subscript
Formatted: Subscript
Formatted: Subscript
Formatted: Subscript
Formatted: Subscript
Formatted: Subscript
Formatted: Subscript
Formatted: Subscript
Formatted: Subscript
Formatted: Subscript
Deleted: Section Break (Continuous)

DTDs were produced using the generator with the input parameters mb and ms both set to 5.

Note that we use letters from the alphabet as subelement symbols.

<table>
<tr><th>No.</th><th>Original DTD</th></tr>
<tr><td>1</td><td>abcde|ef gh|ij|klm</td></tr>
<tr><td>2</td><td>(a|b|c|d|f)* gh</td></tr>
<tr><td>3</td><td>(a|b|c|d)* |e</td></tr>
<tr><td>4</td><td>(abcde)* f</td></tr>
<tr><td>5</td><td>(ab)* |cdef |(ghi)*</td></tr>
<tr><td>6</td><td>abcdef(g|h|i|j)(k|l|m|n|o)</td></tr>
<tr><td>7</td><td>(a|b|c)d* e* (f gh)*</td></tr>
<tr><td>8</td><td>(a|b)(cdefg)* hijklmnopq(r|s)*</td></tr>
<tr><td>9</td><td>(abcd)* |(e|f |g)* |h|(ijklm)*</td></tr>
<tr><td>10</td><td>a* |(b|c|d|e|f)* |gh|(i|j|k)* |(lmn)*</td></tr>
</table>

Formatted Table

Table 7: Synthetic DTD Data Set

The ten synthetic DTDs vary in complexity with later DTDs being more complex than the earlier ones. For instance, DTD 1 does not contain any metacharacters, while DTDs 2

through 5 contain simple sequencing and OR patterns. DTD 6 represents a DTD in factored form while in DTDs 7 through 10, factors are combined with sequencing and OR patterns.

Deleted: Section Break (Continuous)

Real-life DTD Data Set: We obtained the real-life DTDs from the Newspaper Association of America (NAA) Classified Advertising Standards XML DTD produced by the NAA Classified Advertising Standards Task Force. We examined this real-life DTD data and collected six representative DTDs that are shown in Table 8. Of the DTDs shown in the table, the last three DTDs are quite interesting. DTD 4 contains the metacharacter ? in conjunction with the metacharacter *, while DTDs 5 and 6 contain two regular expressions with * 's, one nested within the other.

Deleted: (this can be accessed at http://www.naa.org/technology/clsstdtf/Adex010.dtd).

Formatted Table

No.	Original DTD	Simplified DTD
1	<ENTITY % included-elements "audio-clip|blind-box-reply|graphic|linkpi-char|video-clip">	a|b|c|d|e
2	<ELEMENT communications-contacts (phone|faxjemail|pager|web-page)*>	(a|b|c|d|e)*
3	<ELEMENT employment-services(employment-service.type; employment-service.location * (e.zz-generic-tag)*)>	ab* c*
4	<ENTITY % location"addr* , geographic-area?, city?, state-province?,postal-code?, country?">	a* b?c?d?
5	<ELEMENT transfer-info(transfer-number; (from-to, company-id)+,contact-info)*>	(a(bc)+d)*
6	<ELEMENT real-estate-services(real-estate-service.type, real-estate-service.location?, r-e.response-modes*> r-e.comment?)* ?	(ab?c* d?)*

Table 8: Real-life DTD Data Set

Deleted: ¶

QUALITY OF INFERRED DTDS

Synthetic DTD Data Set: The DTDs inferred by XTRACT and DDbE for the synthetic data set are presented in Table 9. As shown in the table, XTRACT infers each of the original DTDs correctly. In contrast, DDbE computes the accurate DTD for only DTD 1 which is the simplest DTD containing no metacharacters. Even for the simple DTDs 2-5, not only is DDbE unable to correctly deduce the original DTD, but it also infers a DTD that does not encompass the set of input sequences. For instance, one of the input sequences encompassed by DTD 2 is gh which is not encompassed by the DTD inferred by DDbE. Thus, while XTRACT infers a DTD that encompasses all the input sequences, the DTD returned by DDbE may not encompass every input sequence. DTD 4 exemplifies the two typical behaviors of DDbE - (1) sequence f that is not frequently repeated is appended to both the front and the back of the final DTD, and (2) symbols that are repeated frequently are all OR'd together and encapsulated by the metacharacter +. For example, DDbE incorrectly identifies the term (abcde)* to be (a|b|c|d|e)* which is much more general. Thus, the DDbE tool has a tendency to over-generalize when the original DTDs contain regular expressions with * s. This same trend to over-generalize can be seen in DTDs 8-10 also. On the other hand, as is evident from Table 9, this is not the case for XTRACT which correctly infers every one of the original DTDs even for the more complex DTDs 8-10 that contain various combinations of sequencing and OR patterns. This clearly demonstrates the effectiveness of the generalization module in discovering these patterns and the MDL module in selecting these general candidate DTDs as the final DTDs.

Also, as discussed earlier, DDbE is not very good at factoring DTDs. For instance, unlike XTRACT, DDbE is unable to derive the final factored form for DTD 6. Finally,

DDbE infers an extremely complex DTD for the simple DTD 7. The results for the synthetic data set clearly demonstrate the superiority of XTRACT's approach (based on the combination of generalizing, factoring, and selecting using MDL principles) compared to DDbE's for the problem of inferring DTDs.

Real-life DTD Data Set: The DTDs generated by the two algorithms for the real-life data set are shown in Table 10. Of the five DTDs, XTRACT is able to infer all five correctly. In contrast, DDbE is able to derive accurate DTDs only for DTDs 1 and 2, and an approximate DTD for DTD 3. Basically, with an additional factoring step, DDbE could obtain the original DTD for DTD 3. Note, however, that DDbE is unable to infer the simple DTD 4 that contains the metacharacter ?. In contrast, XTRACT is able to deduce this DTD because its factorization step takes into account the identity element "1" and simplifies expressions of the form 1 | a to a?. DTD 5 represents an interesting case where XTRACT is able to mine a DTD containing regular expressions containing nested * s. This is due to the generalization module that iteratively looks for sequencing patterns. On the other hand, DDbE simply over-generalizes the DTD 5 by ORing all the symbols in it and enclosing them within the metacharacter +.

<table>
<tr><th>No.</th><th>Original DTD</th><th>DTD Inferred by XTRACT</th><th>DTD Inferred by DDbE</th></tr>
<tr><td>1</td><td>abcde|ef gh|ij|klm</td><td>abcde|ef gh|ij|klm</td><td>abcde|efgh|ij|klm</td></tr>
<tr><td>2</td><td>(a|b|c|d|f)* gh</td><td>(a|b|c|d|f)* gh</td><td>gh(a|b|c|d|f)+gh</td></tr>
<tr><td>3</td><td>(a|b|c|d)* |e</td><td>(a|b|c|d)* |e</td><td>(e(a|c|d|b)+e)</td></tr>
<tr><td>4</td><td>(abcde)* f</td><td>(abcde)* f</td><td>(f (a|e|d|c|b)+f)</td></tr>
<tr><td>5</td><td>(ab)* |cdef|(ghi)*</td><td>(ab)* |cdef |(ghi)*</td><td>cdef(a|b|g|i|h)+cdef</td></tr>
</table>

<table>
<tr><td>6</td><td>abcdef(g|h|i|j)(k|l|m|n |o)</td><td>abcdef(g|h|i|j)(k|l|m|n|o)</td><td>abcdef(j(o|l|m|n|k)|g(o|l|n| m|k)|h(m|l|n|k|o)|i(o|l|n|m| k))</td></tr>
<tr><td>7</td><td>(a|b|c)d* e* (f gh)*</td><td>(a|b|c)d* e* (f gh)*</td><td>((c|b|a)d+e+|ad+|bd+|c(e+| d+)?|ad*|be*))(f|h|g)+((a| b|c)d+e+|c(e+|d+)?|a(e+|d +)?|b(e+|d+)?)</td></tr>
<tr><td>8</td><td>(a|b)(cdef g)* hijklmnopq(r|s)*</td><td>(a|b)(cdefg)*hijklmnopq(r|s) *</td><td>((((a|b)hijabcdef g)|b|a)(c|g|f |e|d|s|r)+((b|a)?hijkamnop q))</td></tr>
<tr><td>9</td><td>(abcd)* |(e|f |g)* |h|(ijklm)*</td><td>(abcd)* |(i|klm)* |h|(e|f |g)*</td><td>h(a|d|c|b|e|g|f |i|m|l|k|j)+h</td></tr>
<tr><td>10</td><td>a* |(b|c|d|e|f)* |gh|(i|j|k)* | (lmn)*</td><td>a* |(b|c|d|e|f)* |gh|(i|j|k)* | (lmn)*</td><td>(a+|gh)(e|f |d|i|j|l|n|m|k|c|b)+(a+|gh)</td></tr>
</table>

Table 9: DTDs generated by XTRACT and DDbE for Synthetic Data Set

<table>
<tr><th>No.</th><th>Simplified DTD</th><th>DTD Obtained by XTRACT</th><th>DTD obtained by DDbE</th></tr>
<tr><td>1</td><td>a|b|c|d|e</td><td>a|b|c|d|e</td><td>a|b|c|d|e</td></tr>
<tr><td>2</td><td>(a|b|c|d|e)*</td><td>(a|b|c|d|e)*</td><td>(a|b|c|d|e)*</td></tr>
<tr><td>3</td><td>(ab* c*)</td><td>ab* c*</td><td>(ab+c*)|(ac*)</td></tr>
<tr><td>4</td><td>a* b?c?d?</td><td>a* b?c?d?</td><td>(a+b(c|(c?d))?)|((b|a+)?cd)| ((a+|b)?d)|((a+|b)?c)|a+|b)</td></tr>
<tr><td>5</td><td>(a(bc)+d)*</td><td>(a(bc)* d)*</td><td>(a|b|c|d)+</td></tr>
</table>

Deleted: Section Break (Continuous).

Formatted Table

Table 10: DTDs generated by XTRACT and DDbE for Real-life Data Set

The quality of the DTDs inferred by XTRACT was compared with those returned by the IBM alphaworks DDbE (Data Descriptors by Example) DTD extraction tool on synthetic

as well as real-life DTDs. In the experiments, XTRACT outperformed DDbE by a wide margin, and for most DTDs it was able to accurately infer the DTD while DDbE completely failed to do so. A number of the DTDs which were correctly identified by XTRACT were fairly complex and contained factors, metacharacters, and nested regular expression terms. Thus, the results clearly demonstrate the effectiveness of XTRACT's approach that employs generalization and factorization to derive a range of general and concise candidate DTDs, and then uses a selection step preferably comprising minimum descriptor length (MDL) principles as the basis to select from amongst them.

What is claimed is:

1. A document descriptor extraction method comprising the steps of:

generalizing input sequences associated with a document to develop general sequences, said input sequences reflecting the structure of a document;

factoring said input sequences and said general sequences to develop factored sequences;

selecting a document descriptor from said input sequences, said general sequences, and said factored sequences using minimum descriptor length (MDL) principles.

2. The method of claim 1, wherein said selecting step comprises the steps of:

encoding said input sequences, said general sequences, and said factored sequences; and

selecting a document descriptor which encompasses all of said input sequences and exhibits a minimum MDL cost.

3. The method of claim 2, wherein said encoding step <u>employs an algorithm which applies a set of rules</u> comprising:

$seq(D,s) = \varepsilon$ if $D=s$, if D does not contain metacharacters;

$seq(D_1 \ldots D_k, s_1 \ldots s_k) = seq(D_1,s_1) \ldots seq(D_k,s_k)$;

$seq(D_1 | \ldots | D_m,s) = i\ seq(D_i,s)$;

$seq(D^*,s_1 \ldots s_k) = \{k\ seq(D,s_1) \ldots seq(D,s_k)$ if $k>0$; 0 otherwise$\}$;

wherein D is a sequence of symbols, s is a sequence, and i is an index of a regular expression that the corresponding sequence s matches, wherein log m bits are needed to encode index i.

Deleted: Section Break (Continuous)

4. The method of claim 3, wherein said minimum MDL cost is determined by employing an algorithm to solve a facility location problem (FLP), said FLP modified to compute said minimum MDL cost of potential document descriptors.

5. The method of claim 4, wherein said document descriptor is a document type descriptor (DTD), and said document is an eXtensible Markup Language (XML) document.

6. The method of claim 5, wherein said minimum MDL cost comprises summing a first length of bits describing the DTD and a second length of bits for encoding the sequences.

7. A document descriptor extraction method comprising the steps of:

generalizing input sequences to develop general sequences, said input sequences reflecting the structure of data within a document;

selecting a document descriptor from said input sequences and said general sequences using minimum descriptor length (MDL) principles.

8. The method of claim 7, wherein said selecting step comprises the steps of:

encoding said input sequences and said general sequences; and

selecting a document descriptor which encompasses all of said input sequences and exhibits a minimum MDL cost.

9. The method of claim 8, wherein said encoding step employs an algorithms which applies a set of rules comprising:

$seq(D,s) = \varepsilon$ if D=s, if D does not contain metacharacters;

Deleted: Section Break (Continuous)

$seq(D_1 \ldots D_k, s_1 \ldots s_k) = seq(D_1,s_1) \ldots seq(D_k,s_k)$, if D is a concatenation of $D_1 \ldots D_k$;

$seq(D_1 | \ldots | D_m, s) = i\ seq(D_i,s)$;

$seq(D^*, s_1 \ldots s_k) = \{k\ seq(D,s_1) \ldots seq(D,s_k)$ if $k>0$; 0 otherwise$\}$;

wherein D is a sequence of symbols, s is a sequence, and i is an index of a regular expression that the corresponding sequence s matches, wherein log m bits are needed to encode index i.

10. The method of claim 9, wherein said minimum MDL cost is determined by employing an algorithm to solve a facility location problem (FLP), wherein said FLP is modified to compute said minimum MDL cost of potential document descriptors.

11. The method of claim 10, wherein said document descriptor is a document type descriptor (DTD), and said document is an eXtensible Markup Language (XML) document.

12. The method of claim 11, wherein said minimum MDL cost comprises summing a first length of bits describing the DTD and a second length of bits for encoding the sequences.

Deleted: ¶

13. The method of claim 7, further comprising the step of:

factoring said input sequences and said general sequences to develop factored sequences, wherein said factored sequences are available for said step of selecting;

14. A computer-readable medium encoded with a computer program for generalizing input sequences to develop general sequences, said computer program comprising:

a discover OR patterns procedure;

a discover sequence patterns procedure; and

a generalize procedure which calls said discover sequence patterns procedure and calls said discover OR patterns procedure, wherein said discover OR patterns procedure is nested within said discover sequence patterns procedure.

15. The computer-readable medium of claim 14, said computer program further comprising a partition procedure called by said discover OR patterns procedure.

16. A method for generalizing input sequences to develop general sequences comprising the steps of:

discovering OR patterns among said input sequences; and

discovering sequence patterns among said input sequences and OR patterns.

17. The method of claim 16, wherein said step of discovering OR patterns comprises the step of partitioning said input sequences.

Formatted: Indent: First line: 0.5"

Deleted: Section Break (Continuous)

Deleted: program

Deleted: document descriptor extraction

Deleted: of claim 15, utilizing a computer program

Deleted: as set forth in claim 15.¶
¶
17. The method of claim 16,

Deleted: :

Deleted: generalizing said input sequences to create general sequences using said computer program; and¶

18. A document descriptor extraction method comprising the steps of:

generalizing input sequences, said generalizing step comprising the steps of:

discovering OR patterns among said input sequences, and

discovering sequence patterns among said input sequences and OR patterns;

and

selecting a document descriptor from said input sequences and said general sequences.

19. The method of claim 18, wherein said discovering OR patterns step comprises the step of partitioning said input sequences.

20. The method of claim 19, further comprising the steps of:

factoring said input sequences and said general sequences to develop factored sequences, wherein said factored sequences are available to said step of selecting.

21. The method of claim 20, wherein said step of selecting employs minimum descriptor length (MDL) principles.

22. The method of claim 21, wherein said document descriptor is a document type descriptor (DTD) and said document is an eXtensible Markup Language (XML) document.

Deleted: 18.

Deleted: 17

Deleted: 19

Deleted: 18

Deleted: ¶

------Section Break (Continuous)------

20. The method of claim 19, wherein said document descriptor is a document type descriptor (DTD) and said document is an eXtensible Markup Language (XML) document.¶

¶

21. A method for generalizing input sequences to develop general sequences comprising the steps of:¶

discovering OR patterns among said input sequences; and¶

discovering sequence patterns among said input sequences and OR patterns.¶

Deleted: step of discovering OR patterns comprises the step of partitioning said input sequences.¶

¶

23. A document descriptor extraction method, utilizing a method for generalizing input sequences as set forth in claim 22.¶

¶

24. The method of claim 23, further comprising the steps of:¶

generalizing said input sequences to create general sequences using said method for generalizing input sequences; and¶

selecting a document descriptor from said input sequences and said general sequences.¶

¶

25. The method of claim 24, further comprising the steps of:¶

factoring said input sequences and said general sequences to develop factored sequences, wherein said factored sequences are available to said step of selecting.¶

¶

------Section Break (Continuous)------

26. The method of claim 25, wherein said step of selecting employs minimum descriptor length (MDL) principles.¶

¶

27. The method of claim 26, wherein said

PATENT Garofalakis 6-1-36-11-10

TITLE: DOCUMENT DESCRIPTOR EXTRACTION METHOD

ABSTRACT OF THE DISCLOSURE

A document descriptor extraction method and system which creates a document descriptor by generalizing input sequences within a document; factoring the input sequences and generalized input sequences; and selecting a document descriptor from the input sequences, generalized sequences, and factored sequences, preferably using minimum descriptor length (MDL) principles. Novel algorithms are employed to perform the generalizing, factoring, and selecting.

Deleted: The present invention discloses a

Deleted: . The document descriptor extraction method and system

M:\SWeed\Lucent Technologies\23,760\Patent Office\application

Deleted: draft02.wpd